Student experiment
(C course, the second semester)

Software experiment

Creation of multi-task kernel

(ver.2023.12.05: Maintenance of Table of Contents)

Content

Purpose of experiment

This experiment aims to understand the operations regarding basic functions of operating system.

The content of experiment is classified into three categories. The first one is to understand the method of control and data transfer between the programs described in an assembler language and a high-level language. Specifically, the C language is treated as a high-level language. Using C language's library, the environment where the library can be used is constructed on the operating system. It is learned through this work, how to set variable names, to use registers and stacks, and to produce one load module by compiling programs, as a method of calling out a program described in the assembler language from a C language's program. The second ones is to understand the parallel process control system, as one of basic functions to control the operating system. Specifically, the semaphore's function is realized as an operation system. Providing the semaphore's system call, the environment that can be used by plural processes is realized. The function of process switching by the semaphore is also realized as an operation system. The concept of the parallel process control is learned through these works. The last one is to understand the method of synchronous processing by exclusive control. Specifically, using the semaphore's function, it is created the application program attempting the synchronous processing by the exclusive control. Through this work, the basic concept of the exclusive control is learned.

About examinations and reports

During the experiment period, two examinations and report submissions are required. Submit the reports at the same time as the examinations. The date of the first examination is announced separately. The second examination is conducted on the last day of the experiment. Both reports must be prepared by yourself alone. The important points in writing reports are as follows.

Theme 1

The following points are to be noted in the first report (theme 1).

- At least the items below should be written in the report.

Explanation on the theme 1 (It is prohibited copying the textbook word for word.)

Explanation of the positioning of the theme 1 in the whole experiment

Explanation on programs

Programs' list

Important points in your programming

Problems occurred in the programming, their causes and solutions

Discussions

- As a preparation for theme 2, create user task groups satisfying the following two conditions in the multi-task environment with a ready queue, then perform the thought experiment.

- Use more than two semaphores.

- There exists the timing when more than two tasks are coming in all of the semaphore and the ready queue, other than the initial state. (No need to satisfy this condition at the same time in the semaphore and the ready queue.)

In the thought experiment, suppose that the task switching is performed at the place and timing of program, that are not unnatural but convenient to accomplish the experiment. Describe the explanation of the user task programs and their list, and show the operation transition table of temporal changes in the state of the executed task, ready queue and semaphore queue, that are triggered by the switching of time-sharing tasks. Based on these, indicate that the conditions above are satisfied, through the explanation of operation. Examples of operation transition tables can be found in Sections 2.5.7 and 2.5.8.

(Prepare the report, presupposing that it is used for the operation check of the multi-task kernel to be created in theme 2. However, it is not necessary to actually check the operation on the target, because it is difficult to embody the task switching with the convenient timing adopted in the thought experiment.)

Themes 2 and 3

Note the following points in the second report (themes 2 and 3).

- In the part of user task, each of you should create different one, respectively.

- At least the following items should be written for each theme in the report.

Explanation on the theme ((It is prohibited copying the textbook word for word.)

Explanation of the positioning of the theme in the whole experiment

Explanation on programs and program's list

In theme 2, kernel and user tasks, indicate your department in charge and describe your department in more detail than other all part.

In theme 3, modified kernel part and user tasks.

Important points in your programming

Problems occurred in the programming, their causes and solutions

Discussions

- Submission of program files

Each group should put the created program files in a folder with their group number and upload them to the corresponding submission address on the moodle page. The kernel part of

Theme 2 and the task part of Theme 3 (Applications) must both be submitted. The kernel part is one for each group. user Tasks should be submitted by each individual so that it is easy to identify which file belongs to which person. For example, devise file names, or write the correspondence between file names and names in the manifest file.

Before the last day of the experiment, we ask you to fill out a short questionnaire about the experiment on the moodle page.

Chapter 1 Theme 1: Porting of C language library

1.1 Introduction

In the theme 1, the C language processing system, m68k-elf-gcc, is customized so that C language programs can be executed on the target used in the experiment. Specifically, in the library provided in accompany with the m68k-elf-gcc, the read(), write(), printf() and scanf() are ported so that they can be used on the target.

In order to perform this, the following matters should be well understood.

1. **How to create one executable code from the program written in C language and the program written in assembly language.**
2. **How to start a C language program**
3. **How to construct C language's input/output library**

1.2 Basic matters

1.2.1 Separate compilation and assembly of program

In creating a program, it is recommended the method that after the whole program is divided to smaller program files (modules) in accordance with purposes or functions and programs are created for the individual modules, separate compilation/assembly of each module is performed, rather than the method that one huge program is created and its compilation/assembly are performed at once.

The advantageous points for such a separate programming are as follows.

1. **Dividing a work into modules, the task sharing in source program creation by multi-person facilitates better work sharing.**
2. **In the case that a fault occurs on the programming, the cause can be found easily by checking each module.**
3. **The recompilation and reassembly after modifying source program should be performed only for the source program of the module containing a modified part. No need to recompile and reassemble the whole program.**

It is the shortcoming in the separate programming that many small modules are produced for a program and it is time-consuming to compile and assemble them manually. The 'make' command is used as a utility to support this work. Refer to other books regarding the 'make' command. (newline)

It is described in the followings, the matters necessary to perform the C language's separate compilation by the 'm68k-elf-gcc' and the assembly language's separate assembly by the 'm68k-elf-as'.

File A

```
char CommonArray[10];

int func1(int a)
{
    a < b
      ...
}

char func2(char b)
{
      ...
}
```

File B

```
extern int  func1(int  a);
extern char func2(char b);
extern char CommonArray[10];

main()
{
    int  tmp;
    char c;
      ...
    tmp = func1(10);
    c = func2('A');
      ...
}
```

Figure 1.1: File dividing of the C language program

C language program's separate compilation

The C language programs are usually described in one main() function (corresponding to the body in Pascal language) and other functions (functions and procedures in Pascal language). In the Pascal language, the called functions/procedures are described before the calling functions/procedures. In the C language, there are no rules regarding the order of the functions' descriptions. However, in calling function in the C language, it is better to describe the calling functions beforehand. In the case that the function needed to be described after the function call, in source file, only the definition of the function should be described at the beginning of source file.

The C compiler regards one file as a unit of compilation. When compiling one large program by dividing it into several modules, the program is prepared in several separate files, then all of the files are compiled. An example is shown in Figure 1.1. It is described in File A, func1() which is a function to return an int-type value and func2() which is a function to return char-type value, and the function, main(), uses them in File B. Because the C compiler compiles these files separately, it doesn't know the existence of the functions, func1() and func2(), at the time of compilation of File B. In order to inform that these functions are described in another file, the definition of functions are usually written at the beginning of source file. That is the line which begins with 'extern' in File B. The existence of the functions of func1() and func2() is informed to the C compiler by these two lines. Also, in the case of using variables, besides functions, in plural compilation units, the declaration of 'extern' is employed,

def.h

```
#define  NEWLINE   '\n'
#define  TAB       '\t'
#define  NULL      '\0'
#define  FALSE     0
#define  PI        3.1416
```

main.c

```
#include  "def.h"

main()
{
      ...
}
```

Figure 0

like the declaration of array, CommonArray.

As a method to incorporate the content of another file into a source file, there is a method called 'include'. This is the method to incorporate another file by describing in the source file of incorporating side, as follows.

Example: #include "file name"

This designation means that the whole content of the file indicated by "file name" is simply inserted to the place in the program. Though the divided programs are saved in separate files in this method, they are treated as one file in the time of compilation. Therefore, it is prepared a file where the macro used in plural program units are defined, and the file is used so as to be included in each program unit. In Figure 1.2, a macro, such as NEWLINE and TAB, defined in the file "def.h" can be used in the functions in the file of main.c. The file which defines the macro to be used in plural program units, like 'def.h', is called a header file.

In the 'include' above, the file name is surrounded by double quotation marks, but it is surrounded by < and > in some cases, such as #include <file name>. The difference between these two description types is only concerned with which directory the file to be included is searched in. In the case that the file name is surrounded by the double quotations, firstly the file is searched in the current directory, next it is searched in the directory defined by the system. In the case that the file name is surrounded by < and >, the file is searched in the directory defined by the system directly, not searching in the current directory. In the case of including the header file defined by the system, such as stdio.h, it is usually used the one surrounded by < and >.

Separate assembling of assembly language program

Figure 1.3 shows the example of assembly language program (TEST0). Supposing that the subroutine mst in the module TEST0 doesn't use the variables of locala and localb, and consider that the mst is divided to the group (module TEST1) of subroutines foo and bar and variables locala and localb, and the

```
foo:        ...                 mst:        BRA.S   mstbr
        MOVE.L  locala,%D0      mstlp:  ...
        ADD.L   %D0,glob                JSR     foo
        ...                             JSR     bar
        RTS                             ...
                                mstbr:  TST.L   glob
bar:        ...                         BGT     mstlp
        MOVE.L  localb,%D0              RTS
        SUB.L   %D0,glob
        ...                     .section .data
        RTS                     glob:       .dc.l   16
                                locala:  .dc.l   1
                                localb:  .dc.l   2
```

Figure 1.3: Example of assembly language program (before dividing, TEST0)

group (module TEST2) of subroutine mst and variable glob. With dividing the program, it becomes necessary to call the subroutines foo and bar in TEST1 from the subroutine mst in the module TEST2, and to access the variable glob in TEST2 from the subroutine in the module TEST1.

In order to make it possible to use the subroutine and variable in the external module, the .global/.extern directive is employed. Further making the .global declaration of subroutine's label names and variable names (symbol), the declared symbols are given the attribute of external definition and they can be used from the external module. Also, making the .extern declaration in its own module for the symbol with .global declaration in external module, the subroutines and variables in the external

```
        .global foo
        .global bar
        .extern glob
foo:        ...
        MOVE.L  locala,%D0
        ADD.L   %D0,glob
        ...
        RTS
bar:        ...
        MOVE.L  localb,%D0
        SUB.L   %D0,glob
        ...
        RTS

.section .data
locala:  .dc.l   1
localb:  .dc.l   2
```

```
        .extern foo
        .extern bar

mst:        BRA.S   mstbr
mstlp:  ...
        JSR     foo
        JSR     bar
        ...
mstbr:  TST.L   glob
        BGT     mstlp
        RTS

.section .data
.global glob
glob:  .dc.l   16
```

Figure 1.4: Example of dividing assembly language program (Test1(left) and Test2(right))0

module can be used in the own module.

Figure 1.4 shows the example (TEST1 and TEST2) of dividing the assembly language program in Fig. 1.3. Because the module TEST1 needs the variable glob in the module TEST2 and the TEST2 needs the subroutines foo and bar in the TEST1, the .global declaration is made for the symbols foo and bar in the TEST1 and for the symbol glob in the TEST2, and together with this, the .extern declaration is made for glob in the TEST1 and for foo and bar in the TEST2.

The .global/.extern directive can be used from anywhere in the assembly language program. Refer to the manual of m68k-elf-as for details.

Note on split assembly: Valid range of .equ definitions

An .equ string replacement definition written in an assembly language program is valid only within the module. An .equ string replacement definition written in another module (e.g., mon module) is invalid in the currently working module (e.g., inchrw module). If you want to use the same string replacement in the module you are working on as in another module, you need to write a similar string replacement definition by .equ in the module you are working on.

If you want to define a common set of .equ string replacement instructions between modules, you can do the following

1. **Copy-paste the .equ string replacement definition area in the program between modules.**
2. **Create an include file (e.g. equdefs.inc) with the string replacement definitions by .equ and add a line of .include "equdefs.inc" at the top of the assembler program (Figure 1.5).**

The latter is stronger against forgetting to propagate information updates.

```
*** equdefs.
.equ IOBASE, 0x00d00000
** LED
.equ LED7, IOBASE+0x002f ... **
SYSCALL
.equ SYSCALL NUM GETSTRING, 1 ...
```

```
*** inchrw.s
.include "equdefs.inc"
.global inbyte
.text
.even inbyte: ...
        move.l #SYSCALL NUM GETSTRING,%d0 ...
```

Figure 1.5: Use of .include in the assembler programming

1.2.2 Link between assembly language program and C language program

Relation of m68k-elf-gcc/m68k-elf-as/m68k-elf-ld

On this document, the processing system used in the experiment is chosen and explained, from among a variety of C language processing systems and assembly language processing systems. In this experiment, the following three systems are used.

1. **m68k-elf-gcc compiler**
   The source program written in C language is translated to the assembly language source program processable by m68k-elf-as.
2. **m68k-elf-as assembler**
   The source program written in 68000 assembly language is converted to the 68000 object codes processable by m68k-elf-ld.
3. **m68k-elf-ld loader**
   The 68000 object codes are converted to the executable codes. Plural object code files can be linked to single executable code.

Therefore, for example, the following program development can be performed, using these systems.

1. **The single program written in assembly language is converted to the object codes by m68k-elf-as, then converted to the executable codes by m68k-elf-ld.**
2. **The plural programs written in assembly language are converted to object codes by m68k-elf-as respectively, then combined to the single executable code by m68k-elf-ld.**
3. **The single program written in C language is translated to the assembly language program by m68k-elf-gcc. Then this is converted to the object code by m68k-elf-as, and finally to the executable code by m68k-elf-ld.**
4. **The plural programs written in C language are translated to the assembly language programs by m68k-elf-gcc respectively, then these assembly language programs are converted to the object codes by m68k-elf-as respectively, and finally they are combined to the single executable code by m68k-elf-ld.**
5. **The program written in C language is converted to the assembly language program by m68k-elf-gcc. This assembly language program and the program written in assembly language from the beginning are converted to the object codes by m68k-elf-as, respectively, and finally they are combined to the single executable code by m68k-elf-ld. Even in the case of plural C source programs and plural assembly language source programs, the program development can be performed in the same procedure.**

In the simple OS creation in the software experiment I, programs were developed using the process 1 or 2. This time, the program development is to be conducted using the method 5.

Refer to external name

As mentioned in Sec. 1.2.1, in the case that a C language program is divided to plural files and compiled separately, the reference to external names (function name and global variable's name), generated between the files, is converted to an actual subroutine's head address and data storage place on the memory at the time of linking.

Such a reference to external name is clearly indicated by extern declaration in the reference between C source files and by .extern directive in the one between assembly language source file. These declaration and directive are used to show the compiler and assembler that those symbols are defined in the external file. The compiler and assembler reserve in an object file, the information indicating that those symbols should be converted to the actual address at the time of linking.

Then, in the case that a part of program is described by C language and the remaining part is described by assembly language, how should the reference to external names between them be conducted? To conduct this, utilizing the name conversion rule in the occasion that the C program is translated to the assembly language program with keeping the reference to external names, it should be written the extern declaration and .global/.extern directive indicating the external name reference.

In the m68k-elf-gcc used for this experiment, functions and global variables' names in the C program are converted based on the rules below.

- Functions are translated to subroutines. The subroutine's label is the same as the function's name.

- The global variable's definition is translated to the directive to secure the place on the memory, where the global variable is retained. The label is the same as the variable's name.

- The name of function/variable, that is extern-declared, makes the .extern-declaration for the symbol based on the rules described above.

Figure 1.6 shows the example of a simple C program and an assembly language program in the case that the C program was processed by m68k-elf-gcc. (Comments output by the compiler and unnecessary section switching are omitted.)

```
int local;                      .section .text
extern int external;                    .global func2
extern void func1();            func2:
void func2()                            MOVE.L  external, local
{                                       JSR     func1
  local = external;                     RTS
  func1();
}                               .section .bss
                                .extern external
                                .extern func1
                                .global local
                                local:  DCB.B   4,0
```

Figure 1.6: Example of C program and assembly language program 1

Therefore, in the case that the external reference is conducted between s C program and an assembly language program, pay attention to the following points.

1. **Case that assembly language program's subroutine and memory are referred to from C program**
   In the C program, make the extern declaration on the names of functions and variables referred to, and use them as usual. In the assembly language program, write the subroutine and area-securing directive, with the labels for the names of functions and variables used in the C program side, and make the global declaration on these symbols.
2. **Case that C program's function and variable are referred to from assembly language program**
   In the assembly language program side, write the subroutine call and memory reference using symbols, and make the .extern declaration for them. In the C program side, these symbols' names are defined as the names of functions and variables.

Function call

As mentioned in Sec. 1.2.2, a function call is translated to a subroutine call. However, function's argument and return value in C language should also be noted.

How to pass argument

In the m68k-elf-gcc, the argument is generally processed as follows, then the instruction of subroutine call is formed. Refer to the manual, regarding the details on individual data types.

1. **The arguments are evaluated in turn from right to left, and piled up on the stack.**
2. **The data of char (8 bits) and short (16 bits) are sign-extended to 32 bits data using the sign-extending instruction (EXT), then they are piled up on the stack.**
3. **Regarding the arguments (including the structure) other than the array, evaluated values are piled up on the stack.**
4. **Regarding the array, not the copy of the array but its address is piled up on the stack.**

In the subroutine side, these arguments are taken out from the stack and used in the processing. The codes to remove the arguments that became unnecessary after finishing the subroutine, from the stack (to make the stack pointer moved) is put just behind the calling side's subroutine instruction.

At the time when actually writing in assembly language the function called from C language program, firstly write a simple C function with the same arguments range as the called function, then with checking the compiled result for the C function (or with rewriting it), create the assembly language subroutine.

In the case of calling a C function with argument from assembly language program, with checking the compiled result of the called C function, the necessary information should be piled up on the stack, then the subroutine should be called. It could also work, creating another C function only to call the C function and utilizing the result of its compilation.

How to return from function

When returning from an assembly language subroutine corresponding to a C function, the returned value is put in the D0 register in the case that it can be expressed in 32 bits. Otherwise, because there is a case that the calling side prepares the stack frame and designates the head address by A1 register, several variations, such as copying the value to the designated address, can be taken.

Usually, in the case that the assembly language subroutine called from the C has a returned value, the program should be written so that the RTS is performed in the state where the value is stored in the D0 register.

In the case that the details are needed, compile the C function only for transferring arguments, and refer to the assembler source generated by the compiler.

On the contrary, in the case of calling the C function in the assembly language program, the program should be described so as to receive the return value from the D0 register.

How to use register in function's body

Some of C functions compiled by the m68k-elf-gcc are used without saving the values of several registers. Therefore, in the case of calling C functions from an assembly language program, the register's value before the calling may has been destroyed prior to the time of return. If necessary, it should be copied to another register beforehand, or the register's value should be saved explicitly by piling up it on the stack.

In the case of calling an assembly language subroutine as a function, from a C program, it is necessary for the register's value to be restored before returning to the calling side, by piling up the register's value on the stack in the subroutine.

1.3 Customization of C language processing system m68k-elf-gcc [preparation for experiment]

1.3.1 Files to be modified/created for customizing m68k-elf-gcc

In constructing the environment where C language programs can be executed on the target, use the hardware initializing function and the system call implementation such as serial input/output, of the simple OS prepared in the software experiment I. Hereafter, the simple OS is called 'monitor' and the monitor program is called 'mon.s'.

In the system to be created in the experiment from now, the following files attached with the system

should be modified or remade, in order to customize the m68k-elf-gcc as a cross compiler[1] for the specific target.

1. **crt0.s**

   This is the assembly language program to describe the processes such as the initializations of BSS section and stack pointer/heap pointer. It is used to call the function main(), after performing necessary procedures. In this experiment, no need to make the modifications, as it has been prepared in advance.

2. **sbrk.s**

   Assembly language routine corresponding to the functions to manage the heap area (storage area where a program uses in the function malloc(), etc.). This is not used in this experiment.

3. **csys68k.c**

   This is a group of C functions to start the function main() and to process the file opening/closing. The process to echo-back the input from the key board, to the screen, should be written in the function read() in the group. In this experiment, no need to make the modification, as it has been prepared in advance.

4. **inchrw.s**

   This stores the one character input function inbyte() that is the base to constitute all of the input-related functions included in the standard I/O library. Therefore, this function should be modified according to the specifications, in order to customize it to the particular target. This time, this function is realized in assembly language, using the GETSTRING system call of the monitor.

5. **outchr.s**

   This contains the one character output function outbyte() that is the base to constitute all of the output-related functions in the standard I/O library. This time, this function is realized in assembly language, using the PUTSTRING system call of the monitor.

1.3.2 Start of C program in m68k-elf-gcc

In the C program, the function main() is started at the beginning. The procedure to execute the main()'s body of executable program compiled by m68k-elf-gcc is as follows.

1. **The main routine of crt0.s is put at the address 400, and the program execution starts here. This routine initializes the BSS section and the stack pointer/heap pointer.**
2. **The main() starts at the end of the routine described above.**
3. **The _main() is executed by the _main() call inserted at the head of the main() when compiled by C**

---

[1] **Cross compiler is the compiler that can compile a program on the CPU different from the CPU for executing the program. In the case of this experiment, programs are compiled by the laptop PC for the program development, and executed on the target board with the 68000 compatible CPU. Similarly, the assembler used this time is called the cross assembler.**

compiler. The _main() initializes C's standard library. Specifically, it initializes variables and I/O buffer, and opens the standard I/O stream.

4. Thereafter, the main()'s content described by a user is executed.

However, the part of above 1. has been modified and the routine (corresponding part in the mon.s) to initialize the monitor program is called after the BSS section's initialization[2]. Hence, the following modifications are performed, as shown in Fig. 1.7.

- Set the label of 'monitor_begin' at the position of starting the initialization of the program (mon.s).

- Insert 'jmp start' at the end of the monitor program's initializing part.

With those modifications, after the initialization of the BSS section in the crt0.s, the routine to initialize the monitor program is executed, then the program returns to the start label in crt0.s again and the processes of crt0.s are continued. Also, in the mon.s, the 'monitor_begin' needs the .global declaration and the 'start' needs the .extern declaration.

In the case of executing the program written in C, it is recommended creating single executable code, by combining the monitor modified above, the crt0.s and csys68k.c attached with the system, the inchrw.s and outchr.s of your own making, the object codes formed from all of these ones and your C program which contains the main() function. This executable code should be loaded, following the prompt after starting the 68000 system.

To perform this, all of these object files are to be linked, using the option to start the loader or loader command file. These contents have already been described in the Makefile.

```
.extern start
.global monitor_begin
monitor_begin:
        ...
        jmp start
```

Figure 1.7: Modification in mon.s

1.4 Content of creation [content of experiment]

One of the important works to customize the C language processing system as a cross compiler for a specific target is to make input/output functions usable. In the C language, all of the input/output functions have been prepared "outside" of the language processing system, not as language's built-in functions but as library functions. Therefore, the standard input/output library should be implanted so

[2] **Though the head of crt0.s begins with the start label in general, in the crt0.s used in this experiment, the start label is moved after the process to bury the BSS section with 0, and the 'jmp monitor_begin' is inserted just before the label. This is to prevent overwriting by the BSS section's initialization (writing 0 into all of BSS section) in crt0.s, in the case that data were written into the BSS section in the initialization part of the monitor program created in the first semester.**

as to be operated on the target.

In m68k-elf-gcc, the library function's object codes are stored in libc.a. Among these library functions, the ones related to input/output are implemented using the following five functions. Customizing them, all of the input/output functions in m68k-elf-gcc become usable, as they are.

- close() … file closing

- create() … file creation

- open() … file opening

- read() … reading data of an arbitrary number of bytes from a file

- write() … writing data of an arbitrary number of bytes to a file

In the hardware used in this experiment, only the read() and write() utilizing serial port are used.

The read() and write() functions are defined in the csys68k.c file. By checking these functions, it is indicated that the read() and write() are implemented using the functions below, respectively.

- inbyte() … one character input function

- outbyte() … one character output function

Therefore, in order to make the read() and write() for the serial port usable, the two functions above should be implemented, using the system calls of GETSTRING and PUTSTRING in the monitor created in the software experiment I. Through this procedure, the scanf() and printf() can also be used. In C language, one character of alphabet is described in char-type with the length of one byte.

Figure 1.8 shows the module configuration diagram. For example, calling printf() from main(), the printf() calls write(), and the outbyte and PUTSTRING system call are called in turn. Then, through the serial port, a character string is shown in the display. Also, in the figure, file name is indicated in such notation as libc.a, the function of C language is indicated in the notation ending with () such as printf() and assembler function (subroutine) is indicated in the notation ending with ':' such as outbyte:.

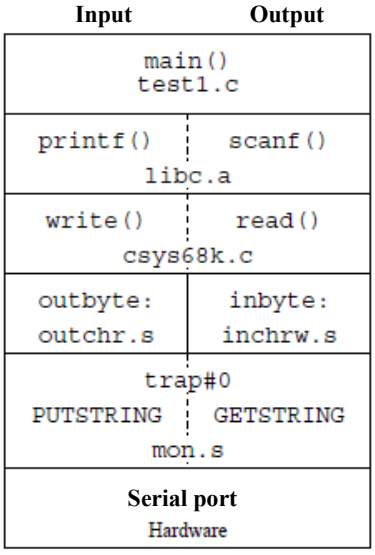

Figure 1.8: Module configuration diagram

1.4.1 Creation of inbyte() in assembly language

The inbyte() is the function to return one character (char-type) read out from the serial port 0 as a return value, without argument. The subroutine with the label of inbyte is to be prepared in assembly language. The monitor's system call GETSTRING is to be called inside the subroutine inbyte. The inbyte() is called from read() in csys68k.c.

Option: Non-waiting 1-character input inkey()

int inkey(int ch); /∗ ch: serial port number ∗/

Create an additional function in inchrw.s. Basically the same as inbyte(), but unlike inbyte(), it does not guarantee a single character input. The return value is an unsigned int value (0x00000000 to 0x000000ff) if a single character is input, or -1 (0xffffffffff) if no single character is input. This is not related to linking with the C library, but may be useful depending on the contents of the application program in Theme 3.

1.4.2 Creation of outbyte() in assembly language

The outbyte() is the function that has char-type data as argument, and outputs the data to the serial port 0. Returned value is not necessarily needed. As mentioned in Sec. 1.2.2, when a function is called, the char-type argument is sign-extended to 32 bits and piled up on the stack. In the subroutine outbyte, the address of the stack storing char-type one byte to be output is set to the register and the monitor's PUTSTRING system call is issued. The outbyte() is called from write() in csys68k.c.

The inbyte() and outbyte() should be the functions inputting/outputting one character without fail. Namely, if the one character output doesn't succeed in the PUTSTRING system call, it should be retried. Not limited to this, in the case of issuing a system call, make sure of the specifications on the system call's input/output and operation. If necessary, the evaluation after issuing the system call should be made.

1.4.3 Compilation and test

Use 'make' command in the compilation. With 'make test1' command, the files of crt0.s, mon.s, inchrw.s, outchr.s, csys68k.c and test1.c are compiled, and the file of test1.abs is formed. The 'make' command recompiles only the updated files. In the case of recompiling all files, execute 'make clean' command first, and then execute 'make test1' command. These operation instructions by 'make' command are described in the 'Makefile' and the Makefile.1, Makefile.2 and Makefile.3 that are called inside the 'Makefile'.

Write and compile a simple C program to perform inputting/outputting, and link it to the modules of monitor, crt0, csys68k, inchrw and outchr, to create one executable code. Loading this using the IPL function of the 68000-side, confirm whether the input/output is performed correctly.

The example of the output in executing the ‘make test1’ is shown below.

```
guest@pcs0xxx% make test1
make LIB JIKKEN=... −f   Makefile.1
make[1]: Entering directory ...
m68k−elf−as −m68000 −ahls=crt0.glis −o crt0.o crt0.s
m68k−elf−as −m68000 −ahls=mon.glis −o mon.o mon.s
m68k−elf−as −m68000 −ahls=inchrw.glis −o inchrw.o inchrw.s
m68k−elf−as −m68000 −ahls=outchr.glis −o outchr.o outchr.s
m68k−elf−gcc −c −g −O2 −Wall −m68000 −msoft−float −I. −Wa,−ahls=csys68k.glis −o csys68k.o
    csys68k.c
m68k−elf−gcc −c −g −O2 −Wall −m68000 −msoft−float −I. −Wa,−ahls=test1.glis −o test1.o test1.c
m68k−elf−ld −nostdlib −−cref crt0.o mon.o inchrw.o outchr.o csys68k.o test1.o −o test1.hex −Map
    test1.map −T /m68k−5.0/lib/soft−jikken/ldscript.cmd perl .../sconv.pl test1.hex > test1.abs
python .../gal.py test1.map crt0.glis mon.glis inchrw.glis outchr.glis csys68k.glis test1.glis >
/dev/null
 make[1]: Leaving directory ...
guest@pcs0xxx%
```

Note 1: Even if the implementation of inchrw and outchr is normally made, the system may not work some cases, depending of the quality of the monitor. Therefore, preparing for the case of not working normally, monitor’s replacement should be added to the items to be considered. The monitor’s replacement can be realized by copying the ‘mon_r.o’ of the distributed file to the ‘mon.o’. (The ‘mon_r.o’ is in the ‘mon r.tgz’, too.) The system call list of ‘mon_r.o’ is described in ‘mon_r.txt’.

Note 2: The use of work field in the monitor at the time of issuing the system call is forbidden. Local work field should be defined in each program, in the sense of improving the prospect of program operation, too. Further, expecting the future multi-task environment, it may become necessary to consider the work field specified for the function calling side. Consider the reason by yourself.

Note 3: Check your understanding of Theme 1 by taking the short exam for Theme 1 on the moodle page before the day of the Theme 1 oral experiment.

1.4.4 Option: Secure a work area equivalent to a local variable area in C function (Difficulty:medium)

Figure 1.9 shows the assembler code (foo.s) generated by compiling (gcc -S foo.c) code that uses local variables of C function. Table 1.1 shows the microcode corresponding to the link and unlk mnemonics that appear in the assembler code.

```
/* foo.c */
int foo(void) {
   int a,b,c;
   a=1;
   b=2;
   c=a+b;
   return c;
}
```

```
** foo.s
   .text
   .even
   .globl foo

foo:
   link.w %a6,#-12       /* a.*/
   move.l %d3,-(%sp)     /* b.*/
   moveq.l #1,%d3
   move.l %d3,-4(%a6) /* c.*/
   moveq.l #2,%d3
   move.l %d3,-8(%a6) /* d.*/
   move.l -4(%a6),%d3
   add.l -8(%a6),%d3     /* e.*/
   move.l %d3,-12(%a6)/* f.*/
   move.l -12(%a6),%d0/* g.*/
   move.l (%sp)+,%d3     /* h.*/
   unlk %a6              /* i.*/
   rts
```

Figure 1.9: Example of a C program and generated assembler code using local variables

Table 1.1: Mnemonic and microcode table of link, unlk

mnemonic	action
link.w %a6,#nnn	#nnn:word value, %sp ← %sp−4, (%sp).l ← %a6, %a6 ← %sp, %sp ← %sp + #nnn
	access by #mmm(%a6) , #mmm ∈ [0,#nnn]
unlk	%a6 ← (%sp).l %sp ← %sp+4

Now observe the assembler code in Figure 1.9

a **link.w %a6,-12 copies %sp to %a6 after pushing %a6.l (4bytes) onto the stack, then updates the value of %sp to -12. Since the updated value is negative, the stack frame area from %sp to less than %a6 is allocated as the working memory area. As in b. immediately below, %sp is used for further stack consumption, but the allocated stack frame area can be accessed with a relative offset reference of %a6. In this case, -4(%a6) is used for int a, -8(%a6) for int b, and -12(%a6) for int c to access the work area.**

b **Since %d3 is destroyed in the code, push %d3 onto the stack in advance.**

c **Corresponds to the a=1 assignment expression in C language.**

d Corresponds to the b=2 assignment expression in C language.

e Keep the calculation result of a+b in C language in %d3

f Corresponds to c=a+b assignment expression in C language.

g Stores the variable value of c in %d0 as the return value.

h Restore destroyed %d3 by popping it off the stack

i unlk %a6 destroys the stack frame area by copying %a6 to %sp, and restores %a6 by popping %a6.l off the stack.

By utilizing the above link and unlk functions, it is possible to safely allocate and use a working memory area from the stack like in C language. Use this mechanism to allocate the necessary work area in inchrw.s.

1.4.5 Option: Add system call number 5 for trap #0 (Difficulty: high)

Consider adding a function to call the timer interrupt processing routine from trap #0 soft interrupt as system call number 5 of trap #0 of the monitor. trap #0 interrupt processing does not save register %d0 in the register save stage because the return value of the system call is returned to %d0. Therefore the register %d0 is not saved at the stage of register saving. On the other hand, in the timer interrupt process, all registers, including %d0, are saved and restored.

Although there is no fixed implementation method, it is recommended to implement the following method, for example:

1) perform a system call number 5 check at the beginning of the trap #0 interrupt processing entry and before saving registers,

2) jmp to the timer interrupt processing entry if a match is found, and

3) shift to the conventional system call check if a match is not found.

If this implementation is used, the timer interrupt processing routine can be called without depending on the implementation of trap #0, even though it is a trap #0 call. The specification for the added functionality of system call number 5 is: %d0=5 on input, no system call return value.

The added functionality of trap #0 system call number 5 is the basis for the implementation of the skipmt() function, which rounds up the allocated time of its own task before a timer interrupt occurs in a timer multitasking environment from Theme 2 onward. Figure 1.10 shows an example of a string definition of the system call number.

```
.equ SYSCALL_NUM_SKIPMT    5
```

Figure 1.10: .equ example of the system call No.5

Chapter 2　Theme 2: Creation of multi-task kernel

2.1 Introduction

The experiment in the theme 1 enabled the program written in C language to be executed on the target.

In the theme 2, plural "jobs" are described as C's functions, and the experiment on multi-task processing is performed, switching and executing the jobs according to the necessity. The multi-task processing is the processing where plural tasks are executed with switching them on one system.

The following cases are considered for the task switching.

1. **In the case that a task uses a shared resource, it is necessary to make the exclusive control in general, and the task switching occurs to realize it.**
2. **In the case that the time for successive execution is limited, the task switching occurs so that only a single task under execution can't occupy the CPU.**

In order to realize task switching caused by these two conditions, the tasks are lined up in two types of queue. One is the ready queue where the task waiting for execution can be restarted any time if the turn for the task comes. The other is the queue for each of shared resources (actually the queue owned by each semaphore, as described later) where the task is in a dormant state because the shared resource is used by another task. The task lined up in the latter queue can't be restarted immediately, even if the turn of using the shared resource comes. It is only woken up from the dormant state and temporally lined up in the ready queue to wait for the execution.

First, it is described the following basic matters needed for task switching, and next, the content of specific processing is explained.

1. **Concept of privileged instruction**
2. **Concept of P/V system call**
3. **Processing necessary to execute with switching a task**
4. **Kernel re-entry prevention by the timing of task switching**

2.2 Basic matters

2.2.1 Concept of privileged instruction

It sometimes becomes necessary to make the distinction for a special work, like that it can be executed In the OS, but can't be executed in a user program. The following two modes are to be set up to make such a distinction.

- **Supervisor mode: All instructions are executable.**
- **User mode: A part of instructions are not executable.**

The instruction that is executable only in the supervisor mode and unexecutable in the user mode is called a privileged instruction. In order to realize such a different execution mode, a bit to indicate the mode is added to computer hardware in many cases. In the MC68VZ328 used for this experiment, the bit for this purpose is prepared in the status register. Refer to the textbook "Development of simple OS" used in the first semester, for the details.

For example, the following instructions are to be executed in the supervisor mode in the operating system. (Hereafter, the sequence of instructions to be executed in the supervisor mode are also called the privileged instruction.)

1. Input/output

All of input/output instructions are the privileged instructions. In the case that the input/output are to be executed in user program side, the system call is executed and the input/output works are requested to the OS operated in the supervisor mode.

2. Set/unset of timer for task switching

The OS has made the setting of timer counter and the setting timer interruption, before passing control to a user program. When the timer generates the interruption, the control is passed to the OS, even if the user program is under operation. Such timer settings should be the privileged instruction.

These operations are realized as the system call using TRAP instruction in the monitor created in the first semester. Accordingly, if these system calls are issued during the execution of task in the user mode, it is switched to the supervisor mode and the necessary processing is performed. Then, returning to the user mode, the task execution can be resumed.

Such a procedure is useful for preventing the serious system breakdown caused by faulty processing in user's task.

2.2.2 Concept of P/V instruction

When two or more tasks try to operate a shared resource simultaneously, the integrity of shared resource is not assured in some cases. Such a resource should be utilized so as not to be accessed from plural tasks simultaneously. The control for this purpose is the exclusive control (mutual exclusion).

Semaphore and P/V instructions using the semaphore have been devised as representative methods to realize this. There are some methods to constitute the semaphore. Here, consider the case that the semaphore is constituted from the counter and queue (list). The counter is set to 1 at the time of initialization. The case of the counter value of 1 indicates there are no tasks using the semaphore (or shared resource managed by the semaphore), the case of the counter value of 0 or less indicates there are tasks using the semaphore and the case of the counter value of less than 0 indicates there are tasks waiting to use the semaphore. Also, the queue is the lineup of tasks waiting to use the semaphore. The P instruction with the semaphore S as an argument is the request to use the semaphore S, and the V

instruction with the semaphore S as an argument is the instruction to release the semaphore S.

When a user task issues the P instruction with a semaphore S as an argument, the counter inside the semaphore is reduced by 1, and the following procedures are conducted according to the resulting values of counter.

- **If the counter value is minus (The value before reduced by 1 is zero or less, so the task using the semaphore exists.), transfer the task to the queue inside the semaphore in order to make the user task go into a dormant state, and switch to another user task.**
- **If the counter value is zero, do nothing. (Namely, continue the user task as it is.)**

On the other hand, when a user task issues the V instruction with S as an argument, the counter inside the semaphore is increased by 1, and the following procedures are conducted according to resulting values of the counter.

- **If the counter value is zero or less (The value before increased by 1 is less than zero, so the task waiting to use the semaphore exists.), take the task out from the semaphore's queue and transfer it to the ready queue.**
- **If the counter value is plus, do nothing.**

For example, suppose that the P instruction is issued to the semaphore S_c corresponding to the shared resource C and then the C is used. Here, if the user task is written so as to issue the V instruction to the S_c when the use of the C is finished, the P instruction can be regarded as the operation to lock the shared resource before it is used, and the V instruction can be regarded as the operation to unlock the shared resource after it is used. Also, in the case that a task A issues the P instruction to the semaphore S and another task B issues the V instruction to the S, the dependency relation in processing, between A and B, can be demonstrated. Namely, the use of the P/V instructions can realize not only the exclusive control but also the switching to another task by interrupting the specific task's execution at a desired time. In this experiment, because there are no shared resources managed by semaphore, the latter one will be attempted.

The P/V instructions are executed with the task switching and the queue operation inside semaphore, so the process on the P/V instructions should be performed in the supervisor mode (namely, as a privileged instruction). Accordingly, the P/V instructions are realized as a system call of OS.

2.2.3 Process necessary to execute with switching task

In the case of conducting multi-task processing, it can be interrupted on the way of executing a statement in the task's program and resumed later. This case, the execution result after resuming the task should be the same as that without switching these tasks. Then, the following two conditions are required.

1. **On interrupting the task's execution, the "state" inside the computer at the moment should be saved (recorded) correctly.**

2. On resuming the task's execution interrupted, the "state" saved in 1 above should be restored and the execution should be restarted just from the interrupted process.

In order to save the state of the task being executed and resume the execution later, at least the following information must be preserved.

- Program counter's value at the moment of interrupting the execution

- Other register's value at the moment of interrupting the execution

- Content of stack at the moment of interrupting the execution

2.2.4 Prevention of kernel re-entry by timing of task switch

The task switch is conducted in the case that the current task enters into the dormant state by the P instruction and in the case of switching by the timer interruption. The P instruction is not issued while the kernel task is running. (The P instruction is issued by a user task.) On the other hand, the timer interruption can be generated while the kernel task is running. This case, the task switching may destroy the consistency of management information for various queues. For example, if the timer interruption occurs and the task switch is conducted just at the moment a kernel task is performing the switching of user task, the information managing the user task can be destroyed.

To avoid this, it is considered the method that in the timer interruption generated while the kernel task is running, the switching of user task is not conducted and the state before the interruption is restored as it is. That is the method adopted by UNIX[3] .

However, in this experiment, this problem is dealt with a simpler measure that the timer interruption generated on the kernel task running should not be accepted. Because the kernel task other than the task switch utilizing the timer interruption is only the processing of the P/V instruction in this experiment, the 'interrupt disabled' (running level 7) is made at the beginning of the assembler routine 'pv_handler' processing the P/V instruction, that is to be mentioned later, and the running level is returned to an original one at the end. However, in the method above, the upper limit of time duration in which the task is executed continuously is not strictly constant. For example, suppose that just at the moment when the continuously executable time is passing out, the task A calls out the putstring system call and the processing is being performed by the kernel (the monitor program). Then the user task switching due to the timer interruption doesn't occur, and the execution of task A is continued again after the system call is finished.

2.3 Outline of task switching in MC68VZ328 [preparation 1]

In the following, the method of multi-task processing on MC68VZ328, the CPU used in this

[3] Different from the measure called 'Timer interrupt disabled' to be described next, this method can record the generation of timer interrupt by temporally accepting the timer interrupt, and it can make the task switching if necessary, checking the record at the time when the kernel task is finished.

experiment, is examined. The followings are the examples of the timing of switching the task executed on the CPU in the case that plural tasks are running in parallel.

1. **The time assigned for the task was run out. => To be waited until the next turn.**
2. **The P instruction was issued, but the semaphore wasn't available. => To be waited until the semaphore becomes available.**
3. **The request for input/output was issued. => to be waited until the input/output is finished.**

Here, the case that the task switching is generated by the factors of 1 and 2 is treated this time. As mentioned in Sec. 2.2.2, the P/V instructions are realized as the system call using TRAP. Also, as described in the previous section, in the case that the user task A's execution is suspended by the timer interrupt and switched to another task, this timer interrupt is the interruption generated during the user task A's running. Therefore, regarding the timer interrupt as the system call pseudoly issued by the user task, it can be realized in mostly the same manner as the P/V instructions. Then, factor 2 is considered in this section.

Suppose that the P instruction is issued during the task A's execution, but it is interrupted (put to sleep) because the semaphore's argument is unavailable. When the semaphore becomes available and the turn to execute the tsk A comes, the original task A should be resumed at (just after) the preceding interruption point. In order to resume, what kind of processing is needed at the time of interruption?

Firstly, check the state change inside the CPU at the time when the interruption is generated. For simplicity, it is assumed that any task controls only the data in the stack and register.

Suppose that the task A executed in user mode is being operated in the state like Figure 2.1, using the user stack. If the interruption is generated in this case, the following procedure is performed automatically inside MC68VZ328.

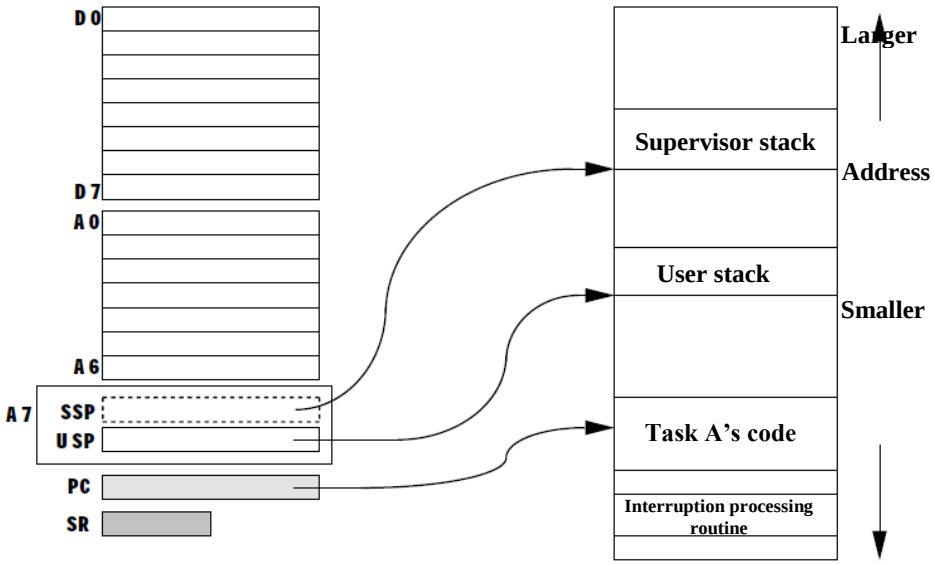

Figure 2.1: State that the user task A is being executed (just before interruption)

1. Shift to supervisor mode

With issuing the TRAP instruction, the MC68VZ328 is shifted to the supervisor mode. Here, the content of status register (SR) is copied to the system register (unaccessible from the program) inside the CPU, and the 13th bit (called S bit) in SR is set to be "1". When the bit is set, the MC68VZ328 is shifted to the state called the supervisor mode. Then, the stack pointer is switched from USP for the user mode to SSP for the supervisor mode.

2. Creation of exception vector's address

Using the number of exception vector, find the address storing the exception vector (interruption processing routine's starting address), and put the address in the system register (different system register from the one where the SR above was copied) inside the CPU.

3. Saving data inside CPU at the time of interrupt generation

Push the value (the address next to the instruction under execution) of program counter (PC) to the supervisor stack. Further, push the value of SR just before the interruption, too. Accordingly, the SSP's value is decreased by 6.

4. Reading out exception vector

Take out the exception vector from the address storing it (stored in the system register by the procedure 2 above), and store it to the PC. The interruption processing is started by the instruction in the address. Figure 2.2 shows the state of system at this point of time.

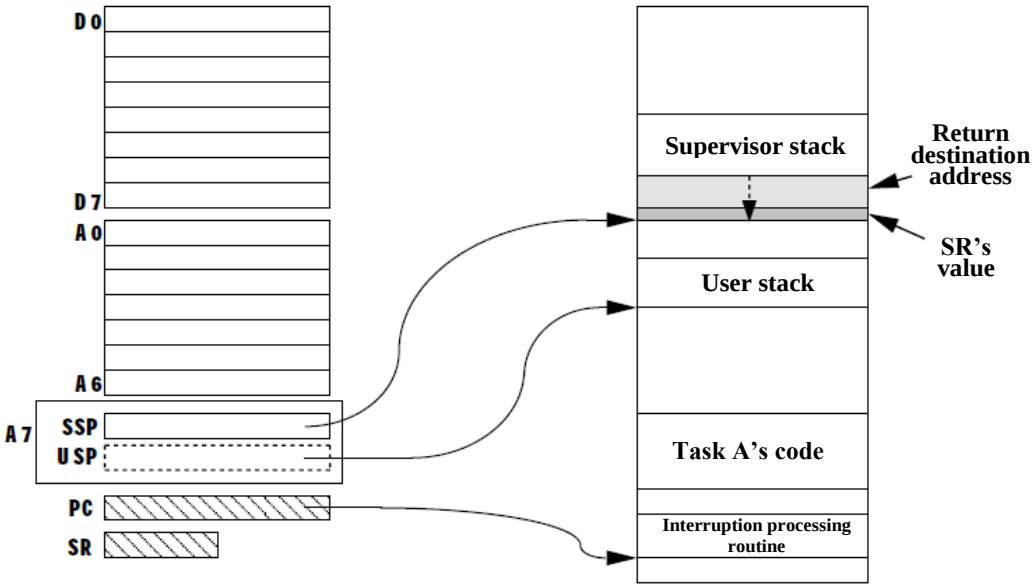

Figure 2.2: State at the start of interruption processing

The body of interruption processing routine is arranged so that its starting address should be the one indicated by the exception vector. At the end of the interruption processing routine, the RTE instruction should be executed without fail.

5. Return from interruption processing

The RTE instruction performs the following processing.

- The data of 2 bytes starting from the position pointed by the SSP is returned to the SR.

- The data of 4 bytes (return destination address) starting from the address of 'SSP + 2' is returned to the PC.

- The SSP is increased by 6.

In the interruption processing above, the supervisor stack is used separately from the user stack and the SR and PC are saved, so the state of user stack and the value of status register program counter are restored successfully.

However, suppose the case that just after the interruption processing generated during the task A's execution, it is resumed without executing other user tasks. Even in this case, the interrupting processing executed in between may destroy the register's value at the interruption occurring time. Further, suppose the case that the execution is actually moved to another user task B after the interruption processing. Depending on the timing that the task B is interrupted and the process returns to the task A, the USP's value and the user stack itself can become different from the state at the time when the task A's execution is suspended, and it is no longer possible to resume the task A's execution with its original state.

The following issues are derived from the considerations above, regarding the task switching in MC68VZ328.

- User stack is necessary for every stack.

Each task has to retain the "state" during its execution. Accordingly, the user stack specified for each task should be set up, and when switching a task, the user stack should also be switched so as not to destroy other tasks' user stacks.

- Supervisor's stack has to retain the values of all registers.

Not only the return destination address of the task being interrupted and the SR's value, but also the values of all registers, have to be recorded in the supervisor's stack used by the interruption processing routine for task switching. This is because it is necessary for all registers to be returned to the state just before the interruption, in order to resume the task in its original state. Also, because the two operations of the switching of user stacks and the saving of all registers' values are not automatically performed by hardware, it has to be carried out by the interruption processing routine for task switching.

- Supervisor stack also has to be prepared for every task at a different position from user stack.

When plural tasks are in the interrupted state, the information (such as register values) necessary to

resume them is different according to each task, so the supervisor stacks are also needed for every task. In the upper part (the part with smaller address) of the supervisor stack just before the switching, it is stored from the top, the USP's value, D0-D7/A0-A's value, return address and SR's value. Thereunder, it is piled up return addresses for the subroutine call performed in the supervisor mode before the task is actually switched. Accordingly, the supervisor stacks can't be prepared continuously from the newest position designated by a user stack pointer, so they should be prepared separately at the position apart from that, to some extent.

- SSP's value for each task has to be recorded separately from stack.

When restoring various kinds of information saved in a supervisor stack for every stack and resuming the interrupted task, the SSP's value after the task was interrupted and each information was saved, has to be obtained. Therefore, the SSP's value at the time when each task is interrupted is needed to be recorded in the position other than the stack.

2.4 Task stack and task control block [preparation 2]

27

It has been understood from the consideration up to the previous section that user stacks and supervisor stacks are needed to be prepared separately for each of tasks operated in parallel in the system created in this experiment. It has also been understood that the part for recording what extent in a stack data are piled up to, is needed separately from the stack.

Then, in the multi-task kernel created in this experiment, the following two kinds of data area are prepared for every task managed.

1. Task stack

It is divided into the user stack's part and the supervisor stack's part. In this experiment, the area with fixed length is allocated to each part at the time of task registration. (The area of 1024 bytes is enough.)

2. Task control block (TCB)

It preserves the task's specific information. The information necessary for the task switching in the theme 2 is only the SSP's value after the saving of register subsequent to the interruption is finished. Besides that, the recording of the following data can be convenient in some cases.

- Address next to the end of user stack (= starting point of supervisor stack)

- Address next to the end of supervisor stack

- Priority of task

- Task's starting address

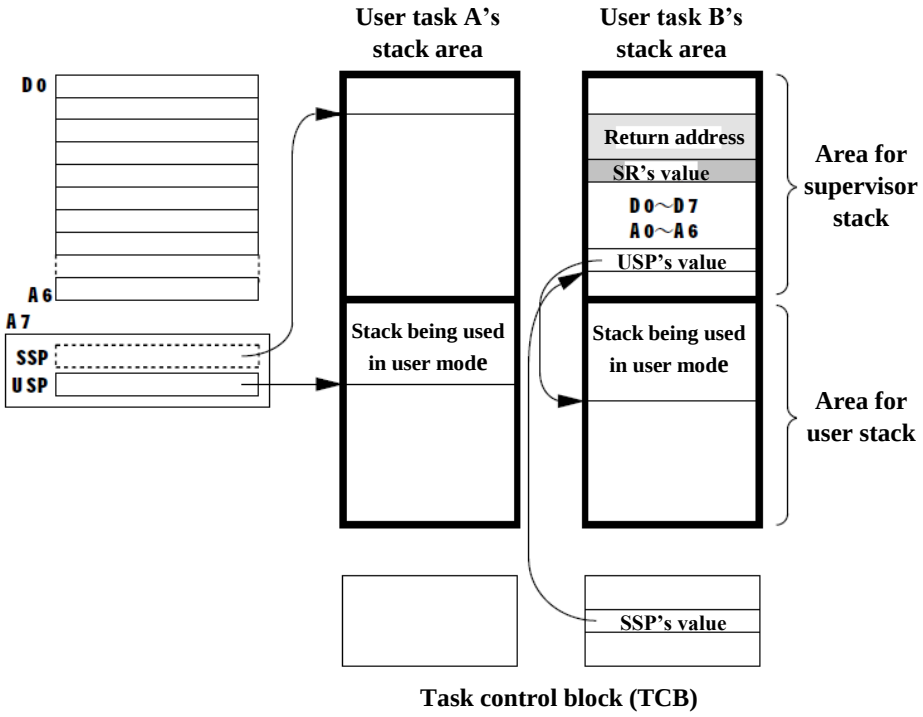

Figure 2.3: Situation of the system under task A's execution

Figure 2.3 shows the schematic chart for these data areas in the cases that the task A in under execution and the task B is under suspension.

In the case that the interruption is generated in the state of Figure 2.3, the address of instruction next to the instruction currently being executed and the SR's value at that time are piled up on the task A's supervisor stack, and the interruption processing is started. The interruption processing routine is operated as follows.

1. **The values of task A's register D0 – D7, A0 – A6 and USP are piled up on the task A's supervisor stack.**
2. **The SSP's value for the task A after the register saving above is finished is registrated inside the task A's TCB.**
3. **Determine the task to be executed next. Here, suppose that the task B is selected.**
4. **From inside the task B's TCB, the SSP's value for the task B is restored. (The supervisor stack is switched from task A to task B.)**
5. **From the supervisor stack for task B, the values of D0 – D7, A0 – A6 and USP are restored. (With the restoration of task B's register group, the user stack is switched to the task B.)**
6. **The RTE instruction is executed.**

The SR's value and the return destination address taken out by the RTE instruction are no longer for the task A, but for the task B. Because the register group and the stack have already been switched to the ones for the task B, the task A's execution is interrupted and the task B's execution is resumed (from the process just after the portion previously interrupted). The stack's state at that time is shown in Fig. 2.4.

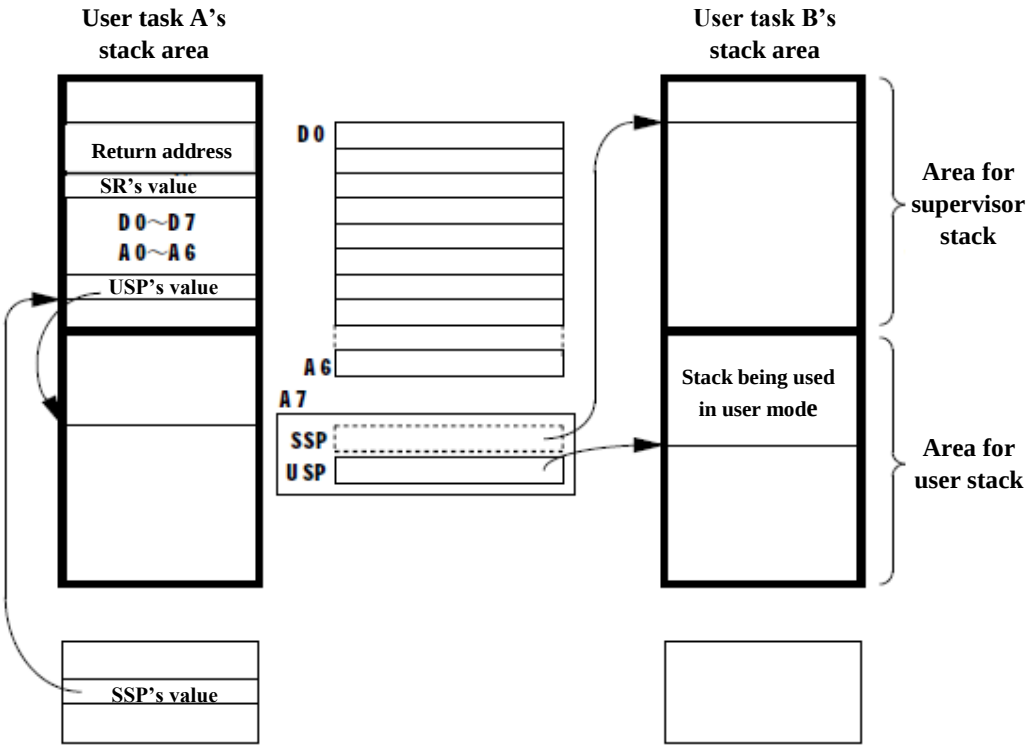

Figure 2.4: Situation after starting task B's execution

2.5 Content of creation [content of experiment]

Figure 2.5 roughly shows the difference in the constitution of main part between ordinary C program and the C program (main part) used in the theme 2's experiment. The two programs show no difference in executing the task1() and task2(), but they differ greatly on the execution form of the process (that is called user task) described in the task1() and task2().

The ordinary program shown in the left of Figure 2.5 calls and executes the functions of the task1() and task2() inside the main(). This case, these functions are executed sequentially from top to bottom, and the switching between the functions is not generated.

On the other hand, the multi-task program for the experiment doesn't call those task functions inside the main(). Instead, it registrates them as a user task using the set_task() function, calls the begin_sch() function and starts the multi-tasking. What is created in the experiment is mainly the C function specific for multi-task processing, such as the set_task() and begin_sch(), and the assembly language routine. These functions are prepared separately from the files shown in Figure 2.5, and linked at the end. In the example above, the headers of these functions are assumed to be included in the "mtk_c.h".

The details about these C functions and the assembler language routine are described later.

```
#include <stdio.h>
…
#define MAX 1024
…
void task1(){
   // task1 definition
}
void task2(){
   // task2 definition
}
void main{
   // hardware initialization
   task1();    // call task1
   task2();    // call task2
}
```

```
#include <stdio.h>
#include “mtk_c.h”
#define MAX 1024
…
void task1(){
   // task1 definition
}
void task2(){
   // task2 definition
}
void main{
   // hardware initialization
   // system setting
   set_task(task1);
   set_task(task2);
   begin_sch(); // start multitasking
}
```

Figure 2.5: Ordinary C program (left) and multi-task experimental C program (right)

2.5.1 Group of files constituting system

The system prepared in this experiment must be realized in the C language and assembly language. Also, what are quite different in nature, like the kernel (the part conducting the task registration and switch) and the user task, coexist in the system. Therefore, they should be prepared as separate files to improve the perspective.

Then, utilizing the function of separate compilation of C language, create the system in the form of separate files as follows. For the convenience of explanation, the names of variables/functions inside the sample program are used, which are allowable to be changed appropriately.

The module constitution chart for the part of multi-task kernel is shown in Figure 2.6. It can be divided broadly to three parts. It is composed of the part to provide the multi-task function, the part to realize the task switching by timer and the part to realize the semaphore, in turn from the left of the chart. The set_timer and reset_timer in the chart are the timer control routine prepared in the software experiment I.

<table>
<tr><td colspan="2"></td><td>MultiTask</td><td>Timer</td><td colspan="2">Semaphore</td></tr>
<tr><td colspan="2">User Mode</td><td></td><td></td><td>P:</td><td>V:</td></tr>
<tr><td colspan="2">Supervisor</td><td rowspan="2">init_timer()
set_task()
begin_sch()
init_stack()
first_task:</td><td rowspan="2">init_timer:

irap #0
set_timer:
reset_timer:</td><td colspan="2">trap #1
pv_handler:</td></tr>
<tr><td colspan="2" rowspan="2"></td><td>p_body()</td><td>v_body()</td></tr>
<tr><td>Queue
operation</td><td>sched() swtch()
addq() removeq()</td><td></td><td>sleep()</td><td>wakeup()</td></tr>
<tr><td colspan="2">HW Interrupt handler</td><td></td><td>hard_clock:</td><td></td><td></td></tr>
</table>

Figure 2.6: Module constitution chart for the part of multi-task kernel

What are to be prepared in C language

1. Main program (“test2.c”) in the experimental system for multi-task processing

It contains the function main(). This is the so-called “main program”, and the following functions are to be started inside it.

- Kernel’s initialization: init kernel();

- User task’s initialization and registration: set_task();

- Start of multi-task processing: begin_sch();

2. Group of user task functions (“user.c” and “user.h”)

The user tasks, task1() and task2(), are described in “user.c”. These user tasks have no arguments and no values to be returned. Prepare them in the form of infinite loop.

The header file “user.h” makes the “extern” declaration for these user task functions, and includes them from “test2.c”. This time, the user task functions’ group can be included in the “test2.c”. That case, the “user.h” is unnecessary.

It is possible not to make the user task function infinite loop. That case, it is necessary to perform the process for ending the user task. Consider this ending process from the two perspectives that the USP points the bottom of the stack at the time of ending the user task and that the ended task is not needed to be executed again. (Attention: If you can not implement the task termination operations, you mast not select to avoid the infinite loop task.)

3. The part of C language in multi-task kernel (“mtk_c.c” and “mtk_c.h”)

This contains the group of functions constituting the kernel. There are following functions among them.

- Kernel’s initialization: init kernel();

- User task’s initialization and registration: set task();

- Initialization of the stack for user task: init stack();

- Start of multi-task processing: begin sch();

- Addition of TCB to the last part of task's queue: addq();

- Removal of TCB from the head of task's queue: removeq();

- Task's schedule function: sched();

- Put a task in a dormant state and perform the task switching: sleep();

- Wake up a task to make it executable: wakeup();

Also, there are following functions that are called from the assembly routine.

- Body of P system call processing: p body();

- Body of V system call processing : p body();.

The header file "mtk_c.h" contains the "extern" declaration of the functions defined out of "mtk_c.c" and "test2.c", and the declaration of global variables related to the kernel. Depending on which of files is included, either of the declaration conducting memory assignment or the external declaration should be used properly. On this occasion, the '#itdef – #else – #endif' can be utilized. Read reference books on C language for the details.

What is prepared in assembly language

This contains the following five assembly routines constituting the kernel, in the part written in assembly language in the multi-task kernel.

1. first_task

Start of the first task … prepared as a function (subroutine) called from the C.

2. pv_handler

Task switching … prepared as an interrupt processing routine. This should not be called as a function, but if it is prepared so as to look like a function from C program, this function's name can be referred as the subroutine's head address inside the C program, so it is easy to be registered in the exception vector.

3. P

Entrance to P system call … prepared as a function (subroutine) called from the C. Putting an appropriate value on the register inside this, execute the 'TRAP #1' instruction.

4. V

Entrance to V system call … prepared as a function (subroutine) called from the C. Putting an appropriate value on the register inside this, execute the 'TRAP #1' instruction.

5. swtch

Function to actually turn on the task switch

6. hard_clock

Routine for clock interruption. Since it is registered using the monitor's system call 'TRAP #0', it

should be written so as to be returned by the rts.

7. init_timer

Routine to register the clock interruption routine 'hard_clock' in the vector table. Utilize the monitor's system call 'TRAP #0'.

2.5.2 Function of kernel described in C language

In this section, regarding the part written in C language in the kernel, its functions are explained. For the convenience of explanation, it is used the names of variables/functions inside the sample system prepared by the experiment supervisors. The names can be changed properly without problem.

Global variable

The global variables necessary in the theme's system are as follows.

1. ID of the task being executed at present: curr_task

2. ID of task being registrated at present: new_task

3. ID of task to be executed next: next_task

4. Queue of task waiting for execution: ready (The type is 'TASK_ID_TYPE' (described later), and it points the first task.)

5. Semaphore's array: semaphore [NUMSEMAPHORE]

6. Task control block's array: task_tab [NUMTASK+1]

7. Task stack's array: stacks [NUMTASK]

The type of task's ID is defined in the C program as follows, and variables are declared in the type of 'TASK_ID_TYPE. The actual type is the int type. The task's ID is a number assigned to a task, and it is numbered as 1, 2, 3, … by the turn of registration. The task with ID of 1 utilizes the task_tab[1] and stacks[0]. Be careful that the indexes for the two arrays of task_tab[] and stacks[] are shifted like this[4] .

```
typedef int TASK_ID_TYPE;
```

The semaphore can be defined, for example, as follows.

```
typedef struct
{
        int             count;
        int             nst;    /*reserved */
        TASK_ID_TYPE    task_list;
} SEMAPHORE_TYPE;
SEMAPHORE_TYPE semaphore [NUMSEMAPHORE];
```

[4] **The task IDs are numbered not as 0, 1, 2, …, but as 1, 2, 3, … This is because the numbering is made after the process numbering in the OS such as Linux and Unix begins from 1. It is not the mistake if the task ID's numbering begins from 0.**

In the queue used in this theme, there are the ready queue and the one which each semaphore has. The task being executed at present has not been registrated in any queue, and it is to be registrated in one of the queues at the time of task switching. Moreover, the task registered in any of queues is not registered in another queue. Accordingly, one queue is registered in one queue at most. Also, the expression like "task is registered in queue" has been employed so far, but the TCB itself can be registered in queue, because the task's information is retained in the TCB. Taking these points into consideration, it is understood that the ready queue and the queue which each semaphore has can be realized using the array, task_tab[] (whose element corresponds to one TCB), defined below.

```
typedef struct
{
        void            (*task addr)();
        void            *stack ptr;
        int             priority;
        int             status;
        TASK_ID_TYPE    next;
} TCB TYPE;
TCB_TYPE        task_tab [NUMTASK+1];
```

As mentioned later, a user task is registrated using the function 'set_task()'. Following the turn of registration, the IDs are given to the tasks as 1, 2, 3, … The information on the user task whose ID is 'id' is memorized in ' task_tab[id]'. Here, the member 'next' indicates the ID of next task in the queue. For example, suppose that there are three semaphores, and each variable's value is given as follows.

```
ready=3
semaphore[0].task list=1
semaphore[1].task list=4
semaphore[2].task list=0

task_tab[1].next=7
task_tab[2].next=5
task_tab[3].next=2
task_tab[4].next=0
task_tab[5].next=0
task_tab[6].next=0
task_tab[7].next=0
```

Then, these variables express the following queues. (Left is the top and right is the end.) Also, the queues are expressed as shown in Figure 2.7.

ready : 3, 2, 5

The queues in semaphore[0] : 1 and 7

The queue in semaphore[1] : 4

The queues in semaphore[2] : empty

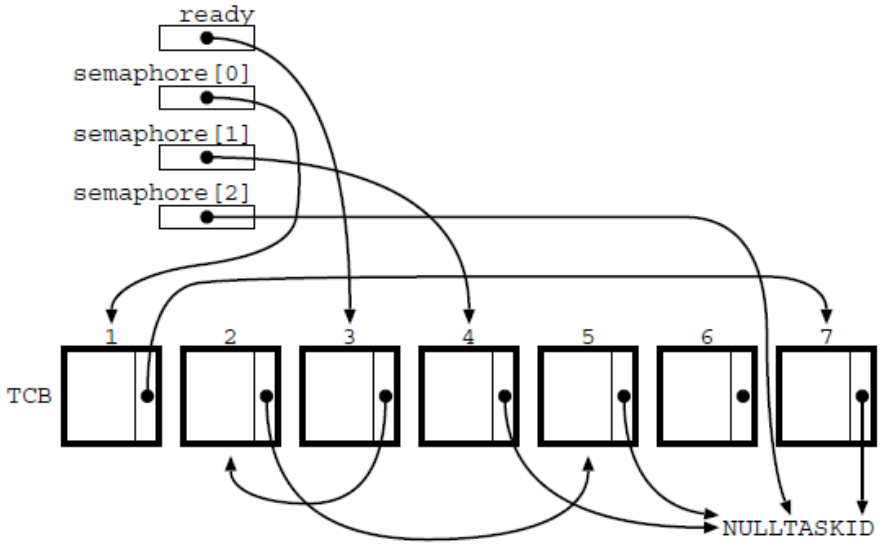

Figure 2.7: TCB and queue

As understood from the description above, the ID0 is used to indicate that it is the end of the queue. (When the values of ready and semaphore's task_list are zero, the queue is empty.)

In the state above, for example, in the case of registrating the task with the ID of 6 at the end of the semaphore[0]'s queue, (after setting the value of the task_tab[6]'s each element, if necessary) search the queue's end (Checking the value 1 of semaphore[0].list, the value 7 of task_tab[1].next and the value 0 of task_tab[7].next, find the task_tab[7] at the end of the queue.), and change the task_tab[7].next to 6. Also, in the case of registrating the task with the ID of 6 at the end of the semaphore[2]'s queue, (after setting the value of the task_tab[6]'s each element, if necessary) search the queue's end in the same way. That case, as the value of semaphore[2].list is zero, this queue is found to be empty and the value of the semaphore[2].list is simply set to be 6. Both cases, the value of the task_tab[6].next is set to be 0 finally, and it is shown that this task is the end of the queue.

For the purpose of making the program easy to see, declare the ID 0 indicating the queue being empty, in "mtk_c.h", as follows. Do not use 0 but use 'NULLTASKID' in the program.

```
#define NULLTASKID   0      /*Queue's termination*/
#define NUMTASK      5      /*Maximum number of tasks*/
```

The NUMTASK that means the maximum number of allowable tasks is treated in the same manner. The NUMTASK of about 5 will be enough in this experiment. As mentioned before, the declaration of TCB's array is made as follows.

```
TCB TYPE          task_tab[NUMTASK+1];
```

The size of array is larger than NUMTASK by 1, because the task's ID begins from 1 and the TCB for the task with ID of 'id' is 'task_tab[id]'.

The 'priority' and the 'status' indicate the task's priority and TCB's usage information (such as 'undefined', 'in use' and 'end of run'), respectively. Though the priority is not necessary in the experiment of theme 2 itself, it may be used in the case of improving the scheduling algorithm afterward. If the task's status is used with declaring appropriate symbolic constants (such as UNDEFINED, for example) in the "mtk_c.h", it will become a program easy to see.

The tsk stack can be defined as follows.

```
typedef struct
{
        char      ustack[STKSIZE];
        char      sstack[STKSIZE];
} STACK TYPE;
STACK TYPE        stacks[NUMTASK];
```

The STKSIZE should be declared in "mtk_c.h". The size of about 1 Kbyte will be enough for this experiment.

Kernel's initialization init_kernel()

No arguments. The following procedure is to be conducted.

1. TCB array's initialization : All elements are made empty.
2. Ready queue's initialization : to be made empty (task ID = 0).
3. Registrate the P/V system call's interruption processing routine (pv_handler) in the interrupt vector of TRAP #1.
4. Initialize the semaphore's value.

Initialization and registration of user task set_task()

Take the pointer (task function's head address) to the user task function as an argument. The following procedure is to be conducted.

1. **Determine the task ID:**

   Find an empty slot in 'task_tab[] (omitting the 0th slot.), and substitute the ID to the 'new_task'.

2. **TCB's update:**

   Register the 'task_addr' and 'status' in the TCB found above.

3. **Stack's initialization:**

   Start the function 'init_stack()'. Register the return value of the function 'init_stack()' in the TCB's 'stack_ptr'.

4. **Registration to queue:**

   Registrate 'new_task' in the ready queue.

In the C language, the array's name means the array's address. The positional information of the stack registrated to the TCB can be indicated using these functions.

Initialization of stack for user task init_stack()

Take the task ID as an argument. The return value is the address (void * type) that the user task SSP points at the time when the initialization is completed. With the argument of id, the following procedure is conducted.

1. **Set the sstack of stacks[id – 1] as shown in Figure 2.8. Set the task's execution starting address 'task_tab[id].task_addr' in the part of "initial PC" in the figure. Set '0x0000' in the part of "initial SR", and skipping over the area of 15x4 bytes, set the user stack top 'stacks[id - 1].ustack [STKSIZE]' in the part of "initial USP".**
2. **Return the address of (*) in Figure 2.8 as a return value.**

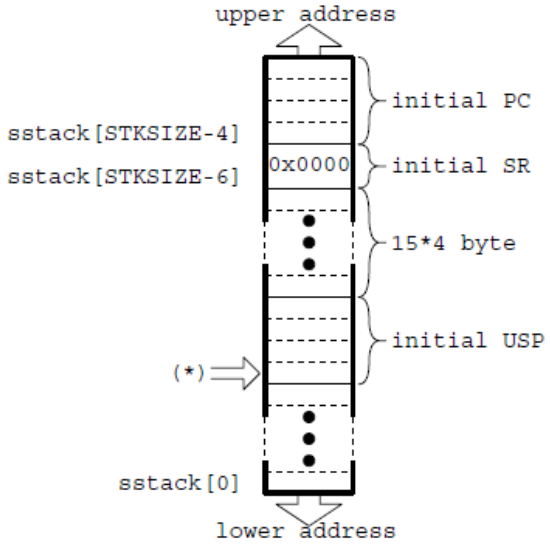

Figure 2.8: Initialization of stack

Furthermore, in the case that the pointer SSP to an int-type is declared, if the ssp's value points the top of the supervisor stack at present, it can be realized to push the value of 4 bytes using '(--ssp) = value'. This makes the operation above easy. Also, when pushing the value of 2 bytes, declare the pointer to an unsigned short int-type, and utilize this.

Start of multi-task processing begin_sch()

No argument. The following processings are conducted.

1. **Determination of first task:**
   Take out one task from the ready queue using removeq() and substitute it for 'curr_task'.
2. **Timer's setting:**
   Call the function init_timer() and set the timer for task switching.
3. **Start of first task:**
   Start the function first_task(), to move the control to the first task. This is converted to the call of assembly language subroutine 'first_task' by the m68k-elf-gcc compiler.

Addition of TCB to the end of task's queue addq()

Take a pointer to queue and a task's ID as arguments, and register the TCB at the end of the queue.

Removal of TCB from head of task's queue removeq()

Take the pointer to queue as an argument, remove the first task from the queue and return the ID.

P system call's body p_body()

The function to conduct the operation of semaphore's value and task list mentioned in Sec. 2.2.2, in P instructions.

1. **Decrease the semaphore's value.**
2. **If the semaphore can't be acquired, execute 'sleep(semaphore's ID)' and enter to the dormant mode.**

V system call's body v_body()

The function to conduct the operation of semaphore's value and task list mentioned in Sec. 2.2.2, in V instructions.

1. **Increase the semaphore's value.**
2. **If the semaphore becomes empty, execute 'wakeup(semaphore's ID)' and make one of the tasks waiting for the semaphore executable.**

Task's schedule function scher()

The task's schedule function (scheduler) is the function that determines the next task to be put in the run state, from the tasks in the ready state, based on the turn that the task was put in the ready state and on the task's priority. This time, because the task's priority is not considered, the task ID at the head of the ready queue should be simply taken out and set to the 'next_task'. Use 'removeq' for the ready queue's operation.

However, in the case that the 'next_task' taken out is 'NULLTASKID', it should be conducted so as to enter into an infinite loop.

Put task in sleeping state and perform task switching sleeo(ch)

1. Connect the current task to the queue of the channel ch (which should be the semaphore ID this case, because only the semaphore is the cause of falling into the sleep state).
2. Start the scheduler 'sched', and set the ID of the task to be executed next to the 'next_task'.
3. Call the task switch function 'swtch' (to be mentioned later) and switch the task.

Make task in sleep state executable wakeup(ch)

Reconnect one task connecting to the queue of the channel ch (which should be the semaphore ID this case, because only the semaphore is the cause of falling into the sleep state), with the ready queue in the executable state. Here, the task switching doesn't occur.

2.5.3 Function of kernel described in assembly language

In this section, the function is explained regarding the part written in assembly language in the kernel. For the convenience of explanation, it is used the names of subroutine labels in the sample system. This can be changed properly.

Subroutine to start user task first_task

This routine is the subroutine that the stack used by the kernel is switched to the stack of the task pointed by 'curr_task', and the multi-task processing is started. This routine is to be activated only once in 'begin_sch(). Because it ends with the RTE instruction, it is not returned to the calling side. At the time of activation, it is needed to be in the supervisor mode.

1. Calculation of TCB's head address:
   Find the address of TCB of 'curr_task'.
2. Restoration of values of USP and SSP:
   Restore the SSP's value recorded in this task's TCB and the USP's value recorded in the supervisor stack.
3. Restoration of all of remained registers:

Restore the values of remained 15 registers piled up on the supervisor's stack.

4. Start of user task:

Execute the RTE instruction.

Entrance of P system call P

Because this subroutine is called with an argument (semaphore ID) from C program, the cautions mentioned in Sec. 1.2.2 are necessary in handling the stack.

Setting the P system call's ID (to be 0) to the D0 register and the argument (semaphore ID) taken from the stack to the D1 register, respectively, execute the 'TRAP #1' instruction. After finishing it, the RTS is to be performed.

Entrance of V system call V

Because this subroutine is called with an argument (semaphore ID) from C program, the cautions mentioned in Sec. 1.2.2 are necessary in handling the stack.

Setting the V system call's ID (to be 1) to the D0 register and the argument (semaphore ID) taken from the stack to the D1 register, respectively, execute the 'TRAP #1' instruction. After finishing it, the RTS is to be performed.

TRAP #1 interrupt processing routine pv_handler

This routine is the interrupt processing routine when the TRAP #1 instruction in P/V of assembler routine was executed, and it is executed in supervisor mode. This routine must not be called from a program in the form of the JSR instruction or the C function 'pv_handler()'.

This routine is called, with the setting the type of P/V system call to the D0 register and the semaphore's ID to the D1 register. According to the D0's value, it is called the C function 'p_body()' or 'v_body(). Because both functions take the semaphore ID (retained in the D1 at the time of interruption) as an argument, it is necessary to make the subroutine call after this is piled up on the stack. The task switching can occur in the p_body().

The outline of the processing is indicated below.

1. Saving of register for task under execution:

Pile up the register used in this routine on the supervisor's stack. Beware that unexpected errors may occur if some register is forgot to be saved.

2. Interrupt inhibition:

After saving the SSR's value to the stack, write '0x2700' into the SR, to set the running level to be 7.

3. Calling out the C's function 'p_body()' or 'v_body()':

Referring to the values of D0 and D1 at the time of interruption, pile up the D1 on the stack as an argument (semaphore ID). If the D0 is 0, call 'p_body()', and if the D0 is 1, call 'v_body()'. The task

switching can occur in the 'p_vody()'.

4. Release of interrupt inhibition:

Restore the SR's value from the stack.

5. Restoration of register:

Restore the register saved at the beginning, from the stack.

6. End of interrupt processing:

Execute the RTE.

Task switching function swtch

This routine is called by the two methods of the 'sleep' and the timer interrupt routine 'hard_clock' (to be described later).

- Case of being called from 'sleep':

It is called in the order of user task, P instruction, TRAP instruction, pv_handler, sleep and swtch.

- Case of being called from 'hard_clock':

It is called in the order of user task, timer interrupt, hard_clock and swtch.

On the other hand, the task to be executed next contains the following task, in addition to the task that has been interrupted by 'sleep' and 'hard_clock'.

- Task executed for the first time since the 'set_task'.

The 'swtch' is called out by JSR, like ordinary C functions, but it executes the RTE when returning finally (and the task switching occurs). This is the measure to treat the task to be executed next in the sane manner, whichever it may be executed for the first time or has already started to be executed. The task that has already started to be executed can return to the calling side ('sleep' or 'hard_clock') by the RTS, but in the case that the task starting for the first time, the 'swtch' should be finished by the RTE in the same way as the 'first_task'.

Then, at the start of 'swtch', the SR's value just after it is called, is piled up on the stack. With this measure, the task that has been executed so far can also be returned to the swtch's calling side. Namely, whichever the task after switching is the one that has been executed so far or it is the one that is executed for the first time, the supervisor's stack for the task becomes USP, D0 – D7/A0 – A7, SR and the return address (the task starting address for the task executed for the first time, otherwise the address inside 'sleep' or 'hard_clock') in turn from the top. Therefore, the process can be conducted so as to restore the USP and D0 – D7/A0 – A7 in turn, and to end with the RTE.

The interrupt process for the task under execution is described through the procedure 3 below, followed by the restoration process for the task to be executed next.

1. The SR is piled up on the stack so that the process can be returned by the RTE.

2. Saving register of task under execution:

The D0 – D7, A0 – A7 and USP are piled up on the task's supervisor stack.

3. SSP's saving:

Find the position of this task's TCB, and record the SSP at the correct position.

4. Change in 'curr_task':

Substitute 'next_task' for 'curr_task'. Because the scheduler 'sched' is started before calling 'swtch', the ID of the task to be executed next is set in the 'next_task'.

5. Read out SSP of next task:

Derive the TCB's position based on the new value of 'curr_task', and restore the SSp's value recorded therein. With this procedure, the supervisor stack is switched to the next task's one.

6. Read out register of next task:

The values of the USP, D0 – D7 and A0 – A7 are restored from the switched supervisor stack.

7. Generate task switching:

Execute the RTE.

Timer interrupt routine hard_clock

This routine is interrupt-driven, but it is returned by the RTS, because it is called from the hardware interrupt processing interface for the timer, that was prepared in the first semester.

1. Saving register of task under execution:

The register used in this routine is piled up on the supervisor stack. This is the routine executed in the timer interrupt, so if some register is forgot to be saved, unexpected errors are generated and the debugging becomes very difficult. Be careful with this processing.

2. Add 'curr_task' to the end of 'ready', using 'addq().

3. Start 'sched'. (The ID of the task to be executed next is set to 'next_task'.)

4. Start 'swtch'.

5. Restoration of register:

The register saved first is restored from the stack.

Clock interrupt routine init_timer

This is created so as to generate the hardware interruption by the timer using the timer control routine prepared in the first semester. The interruption period is arbitrary. For your reference, it is taken to be 10 ms in Linux. The period of about 1 second, that human beings can perceive, is considered appropriate, for the confirmation of task switching operation by the timer.

2.5.4 Compilation and test

The 'make' command is employed in the compilation, in the same manner as Sec. 1.4.3. The files of mon.s, crt0.s, csys68k.c, inchrw.s, mtk asm.s, mtk c.c, outchr.s and test2.c are compiled respectively by the 'make_test2' command, and the 'test2.abs' is formed.

The ‘make test3’ command is almost the same as the ‘make test2’. They are different in the two points that the ‘test3.c is compiled instead of the ‘test2.c’ and that the ‘test3.abs’ is formed.

The example of output for the execution of ‘make test2’ is shown below.

```
guest@pcs0xxx% make test2
make LIB JIKKEN=... −f Makefile.2
make[1]: Entering directory ...
m68k−elf−as −m68000 −ahls=crt0.glis −o crt0.o crt0.s
m68k−elf−as −m68000   −ahls=mon.glis −o mon.o mon.s
m68k−elf−gcc −c −g −O2 −Wall −m68000 −msoft−float −I. −Wa,−ahls=csys68k.glis −o csys68k.o
      csys68k.c
m68k−elf−as −m68000 −ahls=inchrw.glis −o inchrw.o inchrw.s
```

```
m68k−elf−as −m68000 −ahls=mtk asm.glis −o mtk asm.o mtk asm.s
m68k−elf−gcc −c −g −O2 −Wall −m68000 −msoft−float −I. −Wa,−ahls=mtk c.glis −o mtk c.o mtk c.c
m68k−elf−as −m68000 −ahls=outchr.glis −o outchr.o outchr.s
m68k−elf−gcc −c −g −O2 −Wall −m68000 −msoft−float −I. −Wa,−ahls=test2.glis −o test2.o test2.c
m68k−elf−ld −nostdlib −−cref crt0.o mon.o csys68k.o inchrw.o mtk asm.o mtk c.o outchr.o test2.o
      −o test2.hex −Map test2.map −T .../ldscript.cmd
perl .../sconv.pl test2.hex > test2.abs
python .../gal.py test2.map crt0.glis mon.glis csys68k.glis inchrw.glis mtk asm.glis
      mtk c.glis outchr.glis test2.glis > /dev/null
make[1]: Leaving directory ...
```

In order to confirm whether the multi-task kernel is operated correctly, more than two simple user tasks are executed, with switching them alternately. To interrupt the user task’s execution by the P instruction, the semaphore’s value for the argument must be zero or less. Also, to make the task executable later, the V instruction must be issued at least once for the semaphore connected to an inactive task. Utilize the user task for the thought experiment, prepared in the report of the theme 1, or separately create the user task that is simpler and possible to conduct the operation test sufficiently, to perform the operation test.

It is considered that the debugging can be made efficiently with the following steps.

D1: Is the first task operated in the state of the timer interruption being OFF?

The coincidence in stack structure between the task registration (saving) and the restoration of the first task

D2: Is the first task operated in the state that the destination of 'hard_clock' is the RTS only and the timer interruption is ON?

Examination of the influence the timer interrupt processing exerts, and of the monitor replacement

D3: Is the second cycle of task operation generated in the state that the destination of 'hard_clock' is made effective and the timer interruption is ON?

The implementation of 'swtch', storage status of 'curr_task' and operation of ready queue

D4: Confirmation of semaphore's operation, with running semaphore display task under the multi-task environment

Operation status of P, V and trap #1, and control of semaphore's queue

D5: Task switching by semaphore

The use of semaphore as task sequence control

2.5.5 Issues to be noted and others

The issues to be noted in the theme 2 is listed below. Refer to them, if necessary.

1. **Pay attention to the running level during interruption processing.**
2. **The task restoration procedure for 'begin_sch()' and 'swtch:' is conducted in the same manner.**
3. **The Unordinary queue operation without using queue control functions may cause the destruction of queue structure.**
4. **Any tasks other than the one under operation should be put in queues of some sort. Do not put any task out of the queue structure.**
5. **There are files related to the software experiments in folder of the moodle, and ~~the link to~~ there is a file of the 68000 Instruction set (PDF). Regarding the instructions related to the USP's control, search USP through the retrieval in the PDF of xpdf.**
6. **Method to access from C program to absolute address**
   - **Writing to the LED0: *(char *)0x00d00039 = 'A';**
   - **Read the toggle SW's status: tswstat = *(char *)0x00d00041;**

   It is possible to rewrite the interruption vector by this method. However, the pointers to the functions and type conversions are needed.
7. **Note: Before the day of the oral examinations for Theme 2 and 3, please take the small exam for Theme 2 and 3 on the moodle page and check your understanding of Theme 2 and 3.**

2.5.6 Option: Implementation of skipmt() (Difficulty:high)(Required 1.4.5)

After this section, we basically return to individual work.

Implement skipmt: that executes trap #0 at system call number 5 with the same structure as init timer: that executes timer-related system calls internally and can be called as a function from the C language. skipmt: implementation requires additional implementation of trap #0 at system call number 5 (sec.

1.4.5) is required to implement skipmt:.

2.5.7 Application of semaphore

In this experimental system, the count element of the semaphore is initialized between the start of main() and the begin sch() call in test*.c. Semaphores are classified into the following two types according to the initial value of the count element of the SEMAPHORE TYPE structure.

Counting semaphore: initial value of count is 1 or more

Binary semaphore: initial value of count is 0

General semaphores are counting semaphores, and the initial value of count is the number of shared resources. It is responsible for limiting the number of tasks that can pass through a critical section. Binary semaphores, on the other hand, are responsible for stopping and resuming tasks.

While the main application of semaphores is to protect shared resources from allocation contention, another application of semaphore-driven multitasking and task synchronization is shown below. There are two ways to implement semaphore-driven multitasking: using binary semaphores or counting semaphores.

Semaphore-driven multitasking with binary semaphores

Each of the tasks that make up the multitasking environment is matched with a binary semaphore. Assuming that semaphore[k] is mapped to task k (1 ≤ k ≤ N), the implementation policy is shown in Figure 2.9.

```
int task_k(void){

  while(1){
    // some tasks

    P(k);
  }
}
```

a. Step 1

```
int task_k(void){

  while(1){
    // some tasks
    V( (k%N)+1 );
    P(k);
  }
}
```

b. Step 2

```
int task_k(void){
  if (k==N) V(1);
  P(k);
  while(1){
    // some tasks
    V( (k%N)+1 );
    P(k);
  }
}
```

c. Step 3

Figure 2.9: Semaphore-driven multitasking implementation with binary semaphores

Step 1. Task k() is stopped by executing P(k) after task execution in task k().

Step 2. Since there is no opportunity to return the task_k() for step 1, Add to return the task_{k+1}() before stopping the task_k(). In the task_N(), it return the task_{(k%N)+1}().

Step 3. The system starts from the state that task_1,…,task_N() are registered in the ready queue in this order, then each task starts from stopping itself. However, only task N() executes V (1) before P(N).

Each task of this system can't get its own task number, so expand "(k%N)+1" and "if (k==N)" to number and branched code during coding programs. Table 2.1 shows the operation transition table for the case where N = 3.

Table 2.1: Operation transition table for binary semaphore-driven multitasking (N=3)

Events	Curr task	Queue list (Omit task by number)			
		ready	Semp[1]	Semp[2]	Semp[3]
Initialization		1 2 3	-	-	-
begin_sch()	1	2 3	-	-	-
P(1) [task1]	2	3	1	-	-
P(2) [task2]	3	-	1	2	-
V(1) [task3]	3	1	-	2	-
P(3) [task3]	1	-	-	2	3
V(2) [task1]	1	2	-	-	3
P(1) [task1]	2	-	1	-	3
V(3) [task2]	2	3	1	-	-
P(2) [task2]	3	-	1	2	-
V(1) [task3]	3	1	-	2	-
P(3) [task3]	1	-	-	2	3

Semaphore-driven multitasking with counting semaphores

Consider CPU as a shared resource. Figure 2.10 shows an implementation using the counting semaphore semaphore[0].

```
int task_k(void){

   while(1){
      P(0);
      // some tasks
      V(0);
   }
}
```

a. Step 1

```
int task_k(void){
   int f=0; // effective at k=1 only
   while(1){
      P(0); if (f==0){ skipmt(); f=1; } // effective at k=1 only
      // some tasks
      V(0);
   }
}
```

b. Step 2

Figure 2.10: **Semaphore-driven multitasking implementation with a counting semaphore**

Step 1. Place P(0) and V(0) for allocating and releasing CPU before and after the task.

If the processing time per task is longer than the timeout time of the timer, the task will still work as it is.

Step 2. If skipmt() is already implemented, only task_1() executes skipmt() within the P(0)-V(0) interval of the first while loop to cause a forced task switch. Task_2(),…,task_N() cannot allocate

semaphore 0 and enters the semaphore 0 wait queue. In the next turn of task_1(), after V (0), P(0) in the while loop fails to allocate semaphore 0, and task switching occurs to task_2(), which returns with V (0), and after this, semaphore-driven multi-tasking operation is performed.

Table 2.2 shows the operation transition table for the case where N = 3.

Table 2.2: Operation transition table for counting semaphore-driven multitasking (N=3)

Events	Curr task	Queue list		Semp[0] .count
		ready	Semp[0]	
Initialize	-	1 2 3	-	1
begin_sch()	1	2 3	-	1
P(0) [task_1]	1	2 3	-	0
skipmt() or timer before V(0) [task_1]	2	3 1	-	0
P(0) [task_2]	3	1	2	-1
P(0) [task_3]	1	-	2 3	-2
V(0) [task_1]	1	2	3	-1
P(0) [task_1]	2	-	3 1	-2
V(0) [task_2]	2	3	1	-1
P(0) [task_2]	3	-	1 2	-2
V(0) [task_3]	3	1	2	-1
P(0) [task_3]	1	-	2 3	-2

Synchronized task processing

Consider the synchronous processing of task_k() (1 ≤ k ≤ N). Synchronous processing here means that the order in which tasks are stopped due to the completion of the work of each task k is unspecified, but that the tasks synchronize their return to the state after all tasks are stopped. The shared resource variable nttask holds the number of suspended tasks, the counting semaphore (semaphore[0]) protects shared resources, and the binary semaphore (semaphore[1]) suspends tasks. In addition to task_k(), task_0() is prepared for monitoring shared resource variables. Figure 2.11 shows the implementation.

```
volatile int nttask;
int main(void){
   semaphore[0].count=1;
   semaphore[1].count=0;
   nttask=0; ….
   begin_sch();
}
```
a. Step 1

```
int task_k(void){
   while(1){
      // some tasks
      P(0); nttask++; V(0);
      P(1);
   }
}
```
b. Step 2i

```
int task_0(void){
   if (nttask == N){
      nttask=0;
      for (int k=0;k<N;k++)
         V(1);
   }  skipmt();
}
```
c. Step 3

Figure 2.11: Implementation of synchronized task processing by semaphores

Step 1. Initialize with main(). Since shared resource variables are updated by other tasks, it is recommended to use volatile declarations that always use memory values.

Step 2. Task k() is configured to increase the value of nttask by 1 after the end of the task in the loop and to stop at P(1). To protect access conflicts to shared resources, critical sections are set at P(0) and V(0).

Step 3. In the monitoring task task 0(), when the value of nttask matches N, V (1) is executed N times to move N tasks to the ready queue to realize synchronous recovery. Skipmt() frees up more computing resources.

Table 2.3 shows the transition table for the case N = 3.

Table 2.3: Operation transition table for synchronized task operation by semaphore (N=3)

Events	Curr_ task	Queue list		Shared nttask
		ready	Semp[1]	
Initialize	-	0 1 2 3	-	0
begin_sch()	0	1 2 3	-	0
skipmt() [task_0]	1	2 3 0	-	0
P(1) [task_2]	3	0 1	2	1
P(1) [task_3]	0	1	2 3	2
P(1) [task_1]	0	-	2 3 1	3
if (nttask==3) [task_0]	0	-	2 3 1	3
for (k=0,)… [task_0]	0	2 3 1	-	0
skipmt() [task_0]	2	3 1 0	-	0

2.5.8 Option: Implementation of waitP() for synchronized task operation (Difficulty: high)

Figure 2.12 shows an implementation of the function waitP() for inter-task synchronization processing, which is an extension of the semaphore P(1) described in Section 2.5.7, and Table 2.4 shows the operation transition table when N = 3. nst, a reserved element of the SEMAPHORE TYPE structure, is used to store the number of synchronization tasks, N. The number of tasks waiting for synchronization is determined by the count element.

1. Add a second trap#1 and implement waitP: Add a second pv system call (.equ WPCALL, 0x2). Create a function waitP : that calls pv system call number 2 based on P :.
2. Implementation of waitp body() (a.): Create waitp_body() based on

p_body(). The contents are shown in Figure 2.12. Note that a series of v_body() operations when the synchronous processing condition is satisfied (else statement) must be performed before p_body(), which causes a task switch. To improve work efficiency, p_body() should be decomposed and applied each elements to waitp_body().

3. **Initialization (b.) and usage (c.) of waitP(): Semaphores used in waitP() must be initialized to binary semaphores, and the nst element of the semaphore must be initialized with the number of synchronous processing tasks.**

```
void waitp_body(SEMAPHORE_ID_TYPE
        sem_id){
   SEMAPHORE_TYPE *sp;
   sp=semaphore[sem_id];
   if (sp->count != -(sp->nst -1))
      // call p_body()
   else { // decompose p_body()
      for(int k=0;k<sp->nst-1;k++)
         // call v_body()
      // addq ccurr_task to ready
      // call sched(), swtch()
   }
}
```

a. Implementation

```
void main(void){
   …
   semaphore[1]
        .count=0;
   semaphore[1]
        .nst=N;
   …
   begin_sch();
}
```

b. Initialize

```
int task_k(void){
   while(1){
      // some task
      waitP(1);
   }
}
```

c. waitP() usage

Figure 2.12: Implementation of waitp_body()

Table 2.4: **Operation transition table for synchronized task operation by waitP() (N=3)**

Events	Curr_ task	Queue list		Semp[1] .count
		ready	Semp[1]	
Initialize	-	1 2 3	-	0
begin_sch()	1	2 3	-	0
waitP(1) [task_2]	3	1	2	-1
waitP(1) [task_3]	1	-	2 3	-2
waitP(1) entry [task_1]	1	-	2 3	-2
for() v_body [waitP,kernel]	1	2 3	-	0
addq [waitP,kernel]	-	2 3 1	-	0
sched(), swtch() [waitP,kernel]	2	3 1	-	0

Semaphore-related operation tests (other than waitP())

In the final test phase of debug level D5, create and implement programs for semaphore-driven task switching and synchronization processing between tasks by semaphores, as described in section 2.5.7 above, to check their operation.

Execute the operation test of Option:waitP() (Difficulty: high)

Using waitP() implemented based on Section 2.5.8 above, check the operation of inter-task synchronization by semaphore.

Chapter 3　Theme 3 : Applications

After finishing the themes 1 and 2, consider creating the following program using these themes, for example.

1. **Execute three or more tasks in turn.**
2. **Using two RS232C ports, execute the task to perform the output to the port 0 and the task to perform the output to the port 1, in multi-task mode.**
3. **Using two RS232C ports, create plural tasks that receive the input from one port and send some output to another port, and execute them with exclusive control by semaphore.**

Here, in the case of using plural RS232C ports, they can't be used as they are, because the library incorporated in the theme 1 is specialized only for the serial port 0. Therefore, it is necessary to modify the inbyte() and out byte(), and to modify the functions of read() and write() included in the 'csys68k.c.

Make the modification with the attention to the following points, so that two ports can be used properly by the same input/output functions.

- **When calling 'read()' and 'write()' from user task, the input/output ports can be designated by the first argument 'fd'.**
- **The RS232C port 0 is called UART1, and the port 1 is called UART2. Conduct almost the same control for both of them. Refer to the textbook of the software experiment in the first semester, regarding the details of UART1. The comparison between UART1 and UART2 is shown as follows.**
  - **For all of the interrupt mask register (IMR), interrupt status register (ISR) and interrupt pending register (IPR), the Bit 2 corresponds to UART1 and the BIT 12 corresponds to UART2.**
  - **The UART2 also has the register with the same structure as the UART1. While the UART1's register begins from '0xFFF900', the UART2's register begins from '0xFFF910'.**
  - **Interruption vector: The UART2 is in the level 5. (It is possible to be modified. Refer to the MC68VZ328 manual.) The UART1 is in the level 4. (Refer to the textbook in the first semester.)**
- **Because the queue in the monitor (the buffer for character input/output and the control variable (the pointer to the queue's position)) is prepared for the RS232C port 0, the queue for the port 1 is newly copied and to be switched according to the designated port.**

3.1 Stream assignment to RS232C port

In the theme 3, in the case of making both RS232C port 0 and port1 usable, the RS232C's port for input/output object can be designated utilizing the first argument '*fd*' of read() and write(), that has been ignored in these functions up to the theme 2. Instead, the stream input/output finctions, scanf() and printf(), that have been usable up to the theme 2, can't be used, so it is falling into the situation that

there is no way but using the low level functions of read() and write(). Here, use the '*fd*' more appropriately, and make the stream input/output functions usable by conducting the stream assignment.

3.1.1 File descriptor and file pointer

The first argument *fd* of read() and write(), that is called a file descriptor or a file handle, is a non-negative number which is passed as an identifier of input/output operation when opening a file or device using 'open()'. In the C library, the file descriptor's 0, 1 and 2 are assigned to the standard input, standard output and standard error output in advance, respectively. However, this assignment is not necessarily held firmly.

Also, the pointer to the FILE structure, which is passed when opening a file or device using 'fopen()', is called a file pointer. The 'stdin', 'stdout' and 'stderr' are the file pointers assigned to the standard input, standard output and standard error output in advance, respectively. Parameters to control the input/output stream are stored in the FILE structure, and the file descriptor is also stored there.

The 'scanf()' conducts the input operation for the file pointer 'stdin', and the 'printf()' conducts the output operation for the file pointer 'stdout'. Finally, the 'scanf()' is expanded to the 'read()' for the file descriptor 0, and the 'printf()' is expanded to the 'write()' for the file descriptor 1.

Though the file descriptors other than the standard inout/output should be assigned from the kernel by 'open()' and the flag for the 'open()' should also be managed, the 'open()' is not implemented in the experimental kernel, so the simplified implementation is performed within the operatable scope.

As an example, suppose that the file descriptor to the UART1 is assigned to 3, the file descriptor to the UART2 is assigned to 4 and the same file descriptor is to be employed for both input and output. This case, the input/output with the assignment of the UART port is made possible, besides the assignment in the standard input/output.

3.1.2 Use of fdopen() and implementation of fcntl()

In order to make it accessible, the object (RS232C port) that can be input/output by file descriptor, using the stream I/O functions of scanf()/printf(), the I/O stream needs to be assigned to the file descriptor. The 'fdopen()' is the C library function for this purpose.

```
FILE *fdopen(int fd, const char *mode);
```

where '*fd*' is the file descriptor, and '*mode*' is the I/O mode for the *fd*. The "r" is designated as a *mode* in the case of reading, and the "w" in the case of writing. In the case that the assignment for the I/O stream is performed normally, the 'fdopen()' returns a pointer to the file structure other than 'NULL'.

The following global variables are prepared as a pointer to the FILE structure holding the file handle

for the RS232C port.

com0in	**Read out from UART1**
com0out	**Wright into UART1**
com1in	**Read ot from UART2**
com1out	**Write into UART2**

The fdopen(fd, mode) calls out the low level function fcntl(fd, F_GETFL) internally, and confirms the I/O attribute of the file descriptor *fd*. If the attribute obtained can accept the I/O mode designated by the argument of 'fdopen()', it is normal. If not, it gives back the 'EBADF' return value of 'fcntl()' and terminates abnormally. The 'EBADF' is defined by 'errno.h'. Though the 'fdopen()' is implemented inside the C library, the low level function 'fcntl()' is not implemented, so it needs to be newly prepared.

The 'fcntl()' takes the following types.

```
#include <stdarg.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, ...);
```

Because the argument of 'fcntl()' is variable on implementation, it needs to include 'stdarg.h'. In the case of *cmd*=F GETFL, it takes only two arguments (*fd* and *cmd*). In the 'fcntl.h', the values (F_GETFL, O_RDWR, etc.) used inside 'fcntl()' are defined in the 'fcntl.h'.

Regarding inside 'fcntl()', a simplified implementation is performed only in the case that the value of 'cmd' is 'F_GETFL'. Though the flag for '*fd*' at the time of executing 'open()' is usually to be inquired to the kernel, it is not managed in the experimental kernel, so the 'O_RDWR' (I/O available) is always to be returned. In the case that the value of '*cmd*' is not 'F_GETFL', 0 is to be returned simply.

3.1.3 Mapping from file descriptor to real device

The input/output using the file descriptor needs the mapping to the RS232C port number in a low layer. The process up to the theme 2 corresponds to the case that all of I/O designations are mapped to the RS232C port number 0 unconditionally. In the case of this example, the file descriptors of 0, 1, 2 and 3 are mapped to the RS232C port number 0, and the file descriptor 4 is to the RS232C port number 1.

It is adopted the inside of read()/write() functions as a place for the mapping to real devices. The inside of inbyte()/outbyte() functions may be considered, but returning an error is not taken into account in these functions. Also, in the inside of read()/write() functions, the mapping operation can be described in C language, in addition that the determination frequency of device mapping can be decreased.

The determination of device mapping taking the readability/writability into consideration is as follows. The device mapping determination inside 'read()' maps the file descriptors of 0 and 3 to the RS232C port 0, the file descriptor of 4 to the RS232C port 1 and others to error EBADF. The device mapping

determination inside 'write()' maps the file descriptors of 1, 2 and 3 to the RS232C port 0, the file descriptor of 4 to the RS232C port 1 and others to error EBADF.

3.1.4 Summary of works

The 'fcntl()' meeting the specifications above is implemented. Further, using 'fdopen()', create the function that assigns a stream to a file descriptor, and call it at the comparatively early stage of 'main()'. Because the I/O operation for the standard I/O is allocated to the RS232C port 0 with the implementation of device mapping inside the read()/write() functions, the 'scanf()' and 'printf()' can be used in the same manner as the process up to the theme 2. Using the I/O functions of 'fscanf()' and 'fprintf()' that designate file pointers, the input/output designating the RS232C port can be operated equivalently to the 'scanf()' and 'printf()'.

3.2 Serial connection through USB-serial adaptor

At present, the serial port by RS232C is the outdated I/O port (legacy), and the PCs mounting RS23C as a standard have been decreasing. Though some laptop PCs can use the RS23C with extension docks, in the Linux environment the extension docks are not supported in many cases. In order to perform the serial communication by the RS23C under such a situation, the serial connection through a USB-serial adaptor is used in this experiment. The usage is described in the followings.

3.2.1 Connection and device check

Connecting a USB-serial adaptor with a PC's USB port, the recognition and registration of the USB device are conducted in the PC's OS (Linux) side. The result can be checked with the methods below.

1. **Because the log is recorded in '/var/log/messages', execute "tail /var/log/messages" from the command line terminal, and confirm the character string of "pl2303 converter now attached to ttyUSB#". (The '#' is "0" for the first adaptor, and "1" for the second.)**
2. **From the pull down menu, selecting "system => management => system log (system log viewer), confirm the content of '/var/log/messages'. The character string to be confirmed in the log is the same as 1 above.**
3. **Executing "cat /proc/tty/driver/usbserial" from the command line terminal, obtain the table of USB-serial device list. The leftmost number in the table corresponds to '#' of the device file, '/dev/ttyUSB#'.**
4. **Executing "ls /dev/ttyUSB*" from the command line terminal, confirm the existence of the device file, '/dev/ttyUSB#'.**

Confirm the number just after the name of USB-serial device, 'ttyUSB', obtained by the method

described above. (If it is 'ttyUSB0, the number is "0".) It is thought that this number is the turn of recognizing the same device, and not the turn of USB ports.

3.2.2 Start of communication terminal software and its usage

Using the name of USB-serial device type, 'ttyUSB', and its number that have been confirmed in the previous section, start the communication terminal software. If the name of the device is 'ttyUSB0', execute "m68k-termUSB0", and if that is 'ttyUSB1', execute "m68ktermUSB1". The usage is the same as the 'm68k-term' (the name of device is 'ttyS0'), besides that the serial device is different.)

Namely, increase the number of serial port to 2 by the USB-serial adaptor, connect them to real machine's COM1 and COM2 and start the communication terminal software corresponding to serial devices that are different for two command line terminals. Then, one PC can control two serial inputs/outputs. However, because the input can be made only for the command line terminal focused by a mouse, it is recommended the method of connecting two PCs with one real machine, in order to perform the simultaneous input to two serial ports.

Appendix A Exception error message and countermeasure

1. Exception that detailed information is displayed

Bus Error bus time-out decision

Attempted to write into the area of Read-only/Write-protected.

Attempted to access to the area of Supervisor-only in User mode.

Attempted to access to the address space that ROM, RAM, I/O, etc. are mot implemented.

Address Error

Attempted to access to an odd numbered memory address with the command other than '.b'.

Regarding these exceptions, it is separately displayed in addition by the IPL, the 'Access Info' (read/write), the Access Address (address attempted to be accessed) and the IR, SR and PC (the values of Instruction Register, Status Register and Program Counter at the time when exceptions are generated, respectively). They can be used for the information to support debugging. In the case that any cause is not found, it may be the exception described in the item 3 below.

2. Exceptions that can rarely occur

Divide-by-Zero	**Executed the operation of dividing by 0 using DIVS and DIVU instructions.**
Privilege violation	**Attempted to execute the privileged instructions in User mode.**
Spurious Interrupt	**A Bus Error was generated during the interruption processing.**
CHK instruction	**A testing value exceeded the designated range in CHK instruction.**
TRAPV instruction	**Detected an overflow in TRAPV instruction.**
Trace	**Switched to the trace mode with the SR's T-bit of 1.**
Line 1111 emulation	**Attempted to execute the floating point instruction even though there was no FPU.**

Beside the Spurious Interrupt, the exception errors are caused by certain instructions and operations. If the instructions and operations corresponding to these errors have actually been programmed, it is normal stops of the program due to the detection of exceptions, so continue as it is, or stop the instructions and operations that are the cause of error generation. In the case that they have not been actually programmed, move to the item 3 below.

In the case of the Spurious Interrupt, the cause is known to the extent of the Bus error during interruption processing. In the case that the Bus Error is generated simultaneously, also refer to that.

If there seem no applicable ones, move to the item 3 below.

3. Exceptions that do not occur ordinarily

Illegal Instruction Illegal instruction

Line 1010 emulation The pseudo-instruction table of '0xA' is not prepared.

The instructions that were not actually programmed have been executed. It is considered that the address where the program did not exist was stored in the PC, due to the inconsistency of balance in the stacks as a main cause, and the program went out of control,